

# Incremental Transition of C Code into Rust

Bachelor thesis by Philip Koslowski  
Date of submission: October 5, 2020

1. Review: Prof. Dr.-Ing. Mira Mezini (Examiner)
  2. Review: Dr. Krishna Narasimhan (Supervisor)
- Darmstadt



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Computer Science  
Department  
Software Technology

---

## **Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt**

---

Hiermit versichere ich, Philip Koslowski, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 5. Oktober 2020

---

P. Koslowski



---

# Abstract

---

Utilization of C code ranges across countless fields of software development. Nonetheless, C developers have always been dependent on manual memory management, making it nearly impossible to write error-free code, given a certain level of project complexity. Rust introduces a completely new approach, avoiding this issue with its unique concepts that guarantee complete memory and thread safety. Hence, many software companies are considering whether moving their present C code to Rust might be lucrative for them and how this could be achieved. Some automating solutions for C to Rust transpilation are already existent and seem considerable for this purpose at first glance. Furthermore, there are some companies out there, already successfully realizing manual-incremental approaches of C to Rust transitioning.

In this thesis we will show, that usability of present automating C to Rust transpilers has major underlying limitations and that their utilization hardly allows circumnavigation of complete project transpilation, which are hardly ever economically viable for companies. Although manual and incremental solutions are already realized, respective approaches are unapparent and there is no documentation out there sufficing as orientation. Therefore, we will formulate a recipe on the basis of analyzing existent manual-incremental C to Rust project transpilation. Subsequently, we will apply the result on a few open-source projects, showing that it satisfies the requirements of applicability on large and complex software projects.

---

# Zusammenfassung

---

Die Verwendung von C-Code erstreckt sich über unzählige Bereiche der Software-Entwicklung. Nichtsdestotrotz besteht seit jeher das Problem, dass C-Entwickler auf manuelle Speicherverwaltung angewiesen sind, was es ab einem gewissen Grad an Projekt-Komplexität nahezu unmöglich macht fehlerfreien Code zu schreiben. Rust verfolgt einen völlig neuen Ansatz dieses Problem mit seinen einzigartigen Konzepten zu umgehen und ist in der Lage völlige Speicher- und Thread-Sicherheit zu garantieren. Daher stellt sich für viele Software-Unternehmen die Frage, ob sich eine Überführung ihres bestehenden C-Codes nach Rust lohnen würde und wie diese zu bewerkstelligen wäre. Automatisierungs-Lösungen für eine C nach Rust Transpilierung bestehen bereits und sehen auf den ersten Blick nach geeigneten Werkzeugen für dieses Unterfangen aus. Darüber hinaus gibt es einige Unternehmen, die bereits manuelle und inkrementelle Ansätze zur C nach Rust Überführung erfolgreich einsetzen.

In dieser Thesis werden wir zeigen, dass die Nutzbarkeit bestehender C nach Rust Automatisierungs-Software starken Einschränkungen unterliegt und sich durch ihren Einsatz vollständige Projektübersetzungen schwer umgehen lassen, welche meist ökonomisch für Unternehmen nicht tragfähig sind. Manuelle und inkrementelle Lösungen werden zwar schon realisiert, jedoch ist das jeweilige Vorgehen dabei unersichtlich und es besteht keine ausreichende Dokumentation, anhand derer eine Orientierung möglich wäre. Wir werden daher auf Basis der Analyse bestehender manuell-inkrementeller C nach Rust Überführungen ein Rezept erstellen, das Entwicklern als Übersicht dienen kann. Im Anschluss daran werden wir das Ergebnis an einigen Open-Source-Projekten anwenden und zeigen, dass es dem Anspruch für den Einsatz in großen und komplexen Software-Projekten genügt.

---

# Contents

---

<b>1</b>	<b>Introduction</b>	<b>9</b>
<b>2</b>	<b>Motivation and Goals</b>	<b>14</b>
2.1	codecare's Motivation for this Work . . . . .	14
2.2	This Work's Goals . . . . .	15
<b>3</b>	<b>State of the Art Research</b>	<b>16</b>
3.1	C to Rust Tools and Projects . . . . .	17
3.1.1	Automating Tools . . . . .	17
3.1.1.1	Corrode . . . . .	17
3.1.1.2	Citrus . . . . .	18
3.1.1.3	C2Rust . . . . .	19
3.1.1.4	C2Rust Workflow . . . . .	20
3.1.1.5	Assessment of C2Rust's Workflow . . . . .	21
3.1.2	Hybrid Projects . . . . .	22
3.1.2.1	Mozilla Firefox . . . . .	23
3.1.2.2	librsvg . . . . .	25
3.1.2.3	rustybuzz . . . . .	26
3.1.2.4	rust-lzo . . . . .	26
3.1.2.5	raqote . . . . .	26
3.1.2.6	rexpats . . . . .	27
<b>4</b>	<b>Testing Automation with C2Rust</b>	<b>28</b>
4.1	lz4 - Introduction and Data . . . . .	28
4.1.1	Introduction to loc . . . . .	28
4.1.2	lz4: loc . . . . .	29
4.1.3	lz4: Repository Size . . . . .	29
4.1.4	lz4: Tests . . . . .	29

---

4.2	lz4 and C2Rust . . . . .	30
4.2.1	lz4: Transpilation with C2Rust . . . . .	30
4.2.2	lz4: Solving Cargo Build Warnings . . . . .	31
4.2.3	lz4: Assessment of Manual Effort for Functional Equivalency . . . . .	35
<b>5</b>	<b>Rust Background Information</b>	<b>37</b>
5.1	Rust Concepts . . . . .	37
5.1.1	Mutability . . . . .	37
5.1.2	Ownership . . . . .	38
5.1.2.1	References and Borrowing . . . . .	38
5.1.3	Lifetimes . . . . .	38
5.1.3.1	The Borrow Checker . . . . .	39
5.2	Rust Organization . . . . .	39
5.2.1	Cargo . . . . .	39
5.2.2	Modules . . . . .	40
5.2.3	Crates . . . . .	40
5.2.4	Extern Crate Declarations . . . . .	41
5.2.5	Functions . . . . .	41
5.3	C and Rust Interoperability Types . . . . .	43
<b>6</b>	<b>Approaching a Module Porting Recipe</b>	<b>44</b>
6.1	Identifying Modules . . . . .	44
6.2	Replaying librsvg's rsvg-marker Port . . . . .	45
6.2.1	librsvg - Introduction and Data . . . . .	45
6.2.1.1	librsvg: loc . . . . .	46
6.2.1.2	librsvg: Repository Size . . . . .	46
6.2.1.3	librsvg: Tests . . . . .	46
6.2.2	Approaching marker Port Replay . . . . .	47
6.2.3	Filtering Commits . . . . .	47
6.2.4	Gathering Necessary Rust Files . . . . .	48
6.2.5	rsvg-marker.c Removal and rsvg-marker.h Adjustment . . . . .	48
6.2.6	Adjusting Makefile.am . . . . .	48
6.2.7	Enabling Cross-Compiling . . . . .	49
6.2.8	Optional Check for Cargo and rustc . . . . .	49
6.2.9	Optional Debug / Release Switch . . . . .	50
<b>7</b>	<b>Formalization of librsvg's Build Recipe</b>	<b>52</b>

---

---

<b>8</b>	<b>Module Porting to Rust</b>	<b>56</b>
8.1	A Simple Example . . . . .	56
8.1.1	C Implementation . . . . .	57
8.1.2	Porting Adder to Rust . . . . .	59
8.2	calc - Introduction and Data . . . . .	63
8.2.1	calc: Project Structure . . . . .	63
8.2.1.1	calc: loc . . . . .	63
8.2.1.2	calc: Repository Size . . . . .	63
8.2.1.3	calc: Tests . . . . .	64
8.2.2	Dealing with C Pointers in Rust . . . . .	64
8.2.3	Dealing with Constant Values and Static Data in Rust . . . . .	66
8.2.4	Dealing with Multiple if-Statements in Rust . . . . .	70
8.3	The Silver Searcher - Introduction and Data . . . . .	71
8.3.0.1	The Silver Searcher: loc . . . . .	72
8.3.0.2	The Silver Searcher: Repository Size . . . . .	72
8.3.0.3	The Silver Searcher: Tests . . . . .	72
8.3.1	An Overview of rust-bindgen . . . . .	72
8.3.2	Using rust-bindgen . . . . .	76
8.3.3	The Silver Searcher - filename_filter . . . . .	80
8.3.3.1	The Silver Searcher - filename_filter: output . . . . .	80
8.3.4	Dealing with Iterators in Rust . . . . .	82
8.3.5	Dealing with C's Fixed Sized Character Arrays in Rust . . . . .	86
8.3.6	Dealing with Double Pointers in Rust . . . . .	87
8.3.7	Dealing with C's NULL in Rust . . . . .	89
8.3.8	The Silver Searcher: Transpilation Results . . . . .	89
<b>9</b>	<b>Related Work</b>	<b>91</b>
9.1	Is Rust Used Safely by Software Developers? . . . . .	91
9.1.1	Why Does Unsafe Rust exist? . . . . .	91
9.1.2	What Does Unsafe Rust do? . . . . .	93
9.1.3	What Is Analyzed in This Paper? . . . . .	93
9.1.4	How Do the Authors Perform Their Analysis? . . . . .	93
9.1.5	Analysis Results . . . . .	94
9.1.5.1	Popular Libraries . . . . .	94
9.1.5.2	Majority of Unsafe Usage . . . . .	94
9.1.5.3	Perceived and Guaranteed Safety . . . . .	95
9.1.5.4	Is C the Villain Again? . . . . .	95
9.1.6	How Can Rust Uphold Being a Memory-Safe Language? . . . . .	95



9.2	Securing UnSafe Rust Programs with XRust . . . . .	96
<b>10</b>	<b>Conclusion and Future Work</b>	<b>97</b>
<b>11</b>	<b>Data Availability</b>	<b>99</b>
11.1	GitHub Organization: IncrementalTransitionOfCCodeIntoRust . . . . .	99
11.1.1	lz4 Transpilation . . . . .	99
11.1.2	librsvg Marker Port Replay Commits . . . . .	100
11.1.3	calc Transpilation . . . . .	100
11.1.4	The Silver Searcher Transpilation . . . . .	100



---

# 1 Introduction

---

C is everywhere and we cannot live without it. The open-source Linux operating systems for instance are mainly written in C [47], most CPU architectures and operating systems are having a C compiler and it has a stable internal ABI [70]. It manages to be very low-level, which results in very good portability by providing control over memory, while it still can be seen as a high level language, since it is not translated into *machine instructions* directly. However, there are difficulties and challenges with the C language. One such challenge which has critical impact is memory management. Most of the errors produced when writing C are basically the effect of accessing the same memory location multiple times, causing undefined behavior. Some examples are buffer overflows, double frees, use after frees and race conditions [66]. This is hard to avoid, as one needs to think of potential memory related issues beforehand and manually make sure that the chance of their occurrence is minimized.

The following code snippet illustrates an example of how manual memory management can be exploitable, producing a buffer overflow. Imagine a program that checks passwords, having the following implementation in C:

### Example: Exploitable Buffer Overflow in C

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int matches(const char* pw, char* upw, char* msg)
5  {
6      printf(msg);
7      scanf("%s", upw);
8      return strcmp(pw, upw) == 0;
9  }
10
11 int pw_matches(const char* pw, char* upw)
12 {
13     if (matches(pw, upw, "enter password: ")) return 1;
14     else return matches(pw, upw, "wrong, one more try: ");
15 }
16
17 int main()
18 {
19     char user_password[16];
20     const char password[16] = "secretpw";
21
22     pw_matches(password, user_password) ?
23     printf("access granted!\n") :
24     printf("access denied.\n");
25 }
```

The program consists of a *main* function and two helper functions *pw\_matches* and *matches*.

In the main function two character arrays with a fixed length of 16, *user\_password* and *password*, are declared in lines 19 and 20. *user\_password* is uninitialized and mutable, while *password* is *const* and initialized as "secretpw". In line 22 the function *pw\_matches* is called with the previously declared arrays as input parameters, where, depending on whether 1 or 0 is returned, a message is printed, conveying either "access granted!" or "access denied." in lines 23 and 24 respectively.

---

Lines 11 to 15 contain *pw\_matches*' definition. The function takes two parameters. One pointer to a constant character array, *pw*, and one pointer to a mutable character array, *upw*.

In line 13 the function calls its subroutine *matches*, providing two input parameters and a string which informs the caller that he or she shall enter a password.

If this first try is successful, in the sense that the user provided password matches "*secretpw*", the function returns 1, which will print the line "*access granted!*" and exit the program.

In the other case, in line 14, the function *matches* is called once more, informing the caller that his or her password was wrong and he or she has one more try, where now the result returned from *matches* will also be returned from *pw\_matches*, deciding which of the two possibilities is displayed from within *main*.

The function *matches* takes three input parameters, a constant character array, *pw*, a pointer to a mutable character array, *upw*, representing the secret password and password the user will be able to provide, respectively, as well as a pointer to a mutable character array, *msg*, representing the message that is provided via *pw\_matches*.

The function prints out the given message in line 6, scans for the user input, writes it to *upw* in line 7, and compares it with the secret password provided via *pw* in line 8, returning 0 if the two match.

Due to the improper arrangement of the two character arrays *user\_password* and *password*, their addresses are neighbored, such that when *user\_password* experiences a buffer overflow, *password* will get overridden. Once one knows or guesses the length of the arrays, one can exploit the second try to enter the password. Since *password\_usr* can only hold 16 chars, typing "*0123456789abcdefg*" will result in the 'g' overflowing onto *password*, even though it is initialized and const. As *password* is now temporarily overwritten with 'g' for the remainder of the function's execution, we can enter "g" in our second attempt to type in the password, and we will see the message "*access granted!*".

This is something that could not occur in safe Rust, mainly because it does not allow manual management of the respective memory locations. Rust will not let us assign more or less to a 16 character array than 16 characters. Neither is there a way to write user input into an array of predefined length. Rust performs memory allocation after the length of what the input to the program will be is known.

---

The larger the C project gets, the more one approaches the impossibility to completely avoid errors related to memory management. This brings two negative effects. Analysis of the respective issue is hard most of the time, since one needs to trace back all pointer passing, which can get arbitrarily complex. This is where a **debugger** is needed to produce **stack traces**, so one can even get started; but still, analyzing and solving **segmentation faults** in a large project is something that requires a lot a patience and very particular training. The second negative effect is that optimization often suffers from the circumstance that getting to a point where one can be sufficiently positive that a program is free of these errors is hard enough to acquire. Something similar goes for multi-threading, which adds more combinations of ways things can go wrong, like race conditions, again regarding manual memory management. Other difficulties are type and mutability handling. The main reason here, just like with memory management, is that one needs to do it manually. The compiler will help every once in a while, telling what is wrong, but the programmer will have to think of everything he or she wants to get right. Forgetting to make some variable or function *const* can produce security leaks, inferring an unspecialised type can cause ambiguities, to only name some examples.

### **What makes Rust eligible as a C replacement?**

Rust is a programming language invented by Mozilla. The original aim was to close security gaps and perform optimizations in their browser, Mozilla Firefox. Interestingly, the idea was to incrementally replace C and C++ from the browser's code with Rust, which already gives the impression that the language was not invented with the intention of having to rewrite all present code, at least not at once. Rust is addressing many of the discussed issues one encounters while writing C code. While an argument could be that C++ is already doing this for C, one does get improvements, but still many of the addressed C issues remain unsolved in C++. The way Rust addresses these issues is way more drastically. It does not build upon C, trying to solve problems via extensions to an already existing language, but arranges things completely different from the ground up. Rust brings absolute memory safety due to its unique concept of **mutability**, **ownership** and **lifetimes**. As far as mutability is concerned, Rust simply takes the opposite route of C, having all variables immutable unless explicitly stated differently. Rust has no **garbage collector**, but a **borrow checker**, keeping track of the scope in which ownership can be "borrowed". Since garbage collectors can cause hanging and pausing, this concept is very well suited for low-level languages, whose precision and predictability one strictly needs to rely on. Most impressively, Rust compiling guarantees thread safety, even without compromising a C++-like zero-cost abstraction philosophy. The concept of lifetimes guarantees that the programmer is informed whenever variable

---

lifetimes are not limited, which relieves the need to think about freeing resources by ourselves, like we need to in C. [66]

There might be cases in which one might argue with the compiler's strict enforcement of Rust's safety guarantees. It is not devious that this situation might occur once in a while, since the Rust compiler will "reject some valid programs rather than accept some invalid programs" [85]. When the programmer is confident that he or she found a case where the Rust compiler is too strict, the `unsafe` keyword can be used to open a block in which Rust's safety checks are defied, without compilation to fail. Other reasons for doing this are interactions with the operating system or even writing an own operating system [85]. Needless to say, most of the time this should not be desired. When constantly wanting to use unsafe Rust, probably choosing Rust in the first place is questionable.

---

## 2 Motivation and Goals

---

---

### 2.1 codecare's Motivation for this Work

---

*codecare software GmbH* is a German company that develops, modernizes and conducts tailor-made software [20]. Since February 2019 I work for them as a student employee. Recently they assigned me the task of finding out more about Rust and the opportunities this rather young language could offer for a company like theirs. As my initial research turned out to be of interest for them, we decided that this topic would be well suited as subject for this thesis.

To support the claim of real-world relevance of incrementally porting C code into Rust, they formulated their motivation as follows.

At codecare we are continuously searching for efficient software modernisation solutions. For us, efficient solutions balance long-lasting technical sustainability and economic expenditure. In most cases, this ultimately depends on source-code maintainability, stability and testability.

Beyond these aspects, we consider C to be inherently "broken" when it is applied to large and complex projects. Memory issues are created far too easily and their avoidance requires enormous concentration, if not brilliancy, from developers. Besides the challenge of writing correct C code, reading is time consuming and requires great efforts even to understand which project critical problem the code addresses. This means C is a language with a high barrier for maintainance, and consequently a risk for any software modernisation strategy.

On the other hand we realise that legacy system renewal is hardly ever economically viable via a "big bang" approach in which the legacy system is completely replaced all at once. Therefore we especially consider those solutions which allow

---

---

partial software modernisation in crucial locations in a localised manner. Due to the challenges of the C programming language we are particularly interested in trialling such a localised approach with regards to modernising C with Rust.

---

## 2.2 This Work's Goals

---

This work shall give an overview about basic and advanced concepts of Rust. A comparison of Rust and C features shall answer the question how present issues in well-known and established programming languages are solved or avoided by modern ones. The work will also address more practice-oriented questions like how difficult it is to port an existing C program to Rust and how well this can be done incrementally. For observing the theory in practice, the codecare Software GmbH will support me in experimenting on open-source projects, which can substantiate this work's proposals.

---

## 3 State of the Art Research

---

There are many IDEs like *eclipse*, *Code::Blocks*, *Visual Studio (Code)* out there, making C preprocessing, compilation and linking more convenient. More importantly, all the mentioned and many more provide support for debugging C programs. What still makes using debugging tools hard is the general approach with C. Debugging a C program means *tracing back* potential errors that are thrown during runtime. In addition to producing stack traces with *glibc* or *libbacktrace*, using debuggers like *gdb* in some IDEs provides the possibility to step through the program execution and show the momentary state of variables, as well as seeing the sequence of actions that led to the error. Nonetheless, identifying the error-prone spots in a given program and understanding the reason for error-occurrences still means much hard, analytic work. This is where Rust takes a completely different route, basically saying "why do we even allow writing code that can inherit errors?". Its guarantees of memory and thread safety completely obviate debugging of memory related issues and certainly are the main reasons why switching from C to Rust is a consideration that companies appear to make with increasing frequency.

Porting or incrementally transferring existing C code to modern languages like Rust is something that some companies are already doing. As stated before, Rust was initially invented for improving Mozilla's Firefox browser and Microsoft recently wrote a COM<sup>1</sup> library in Rust [76]. In addition, even moving other modern languages into Rust seems to become a consideration for larger companies more often. For example there is an interesting article explaining why moving the Discord *Read States* service from Go<sup>2</sup> to Rust made a huge positive boost for them [97]. To this end, there are already some tools out there for C to Rust conversion [9]. One can also find sources that already document

---

<sup>1</sup>Component Object Model (COM) is a binary-interface standard for software components introduced by Microsoft in 1993. It is used to enable inter-process communication object creation in a large range of programming languages. COM is the basis for several other Microsoft technologies and frameworks, including OLE, OLE Automation, Browser Helper Object, ActiveX, COM+, DCOM, the Windows shell, DirectX, UMDF and Windows Runtime. [21]

<sup>2</sup>Google's version of a programming language trying to solve issues in C [34]



---

“lessons learned” [68], having performed such a porting process. Still, depending on the application, the issues addressed there could be narrow cases, and there certainly is a lot to learn and to look out for that has not already been discussed or proven.

---

## 3.1 C to Rust Tools and Projects

---

In this section, we will examine major automating tools for transpiling<sup>3</sup> C to Rust. Subsequently, there will be a listing of projects which already include some C to Rust transitioning progress, providing a brief summary of their purpose and transpilation approach.

### 3.1.1 Automating Tools

When thinking about transition from one programming language into another, the question of whether automation is feasible, arises immediately. Indeed, there are some projects out there which attempt to perform automated transpiling of C code into Rust, but when looking closer, the aspect of up-to-date relevance can actually narrow it down to one tool of greater avail for this purpose. Firstly though, the major ones to the best of my knowledge are *Corrode* [22], *Citrus* [16] and *C2Rust* [9].

#### 3.1.1.1 Corrode

In a talk at the *RustConf 2018* [74], Per Larsen, Co-Founder at *Immunant Inc.* [37], summarizes their respective approaches and underlines advantages of *C2Rust*. Jamey Sharp invented *Corrode*, which Per Larsen points out as

impressive effort for one guy

but also criticizes that it uses the Haskell C Parsing Library, which was

[...] less used, [...] less maintained and battle tested than the Clang C compiler

which resulted in

---

<sup>3</sup>A [...] transpiler is a type of translator that takes the source code of a program written in a programming language as its input and produces an equivalent source code in the same or a different programming language. [78]

---

some limitations.

Without having to dig much deeper into what he meant by those limitations, we can see from Corrode’s GitHub page, that the last commit was made more than 3 years ago, as of writing this thesis, on the 13th of April 2017. Hence, we can assume that the project will be no longer maintained. In Corrode’s GitHub Readme.md, we find an explanation (implicitly) stating “What Corrode is not”, reinforcing this impression:

A Rust module that exactly captures the semantics of a C source file is a Rust module that doesn’t look very much like Rust [...]. I would like to build a companion tool which rewrites parts of a valid Rust program in ways that have the same result but make use of Rust idioms. I think it should be separate from this tool because I expect it to be useful for other folks, not just users of Corrode. I propose to call that program "idiomatic", and I think it should be written in Rust using the Rust AST from `syntex_syntax`<sup>a</sup>.

<sup>a</sup>`syntex_syntax` is a Rust parser and macro expander [23].

That being said, one can condense that this tool is deprecated and, during active maintenance, did not accomplish a status that the author himself would be satisfied with, in terms of being utilizable as a mature C to Rust transpiler.

*Corrode* is a tool that one can classify as a forerunner in the field of automated C to Rust transpilation. It has paved the way for future tools and currently actively maintained ones like *C2Rust*, but has been abandoned at a stage in which it did not accomplish major maturity.

### 3.1.1.2 Citrus

While we can read the subtext, that Corrode’s biggest and not overcome challenge was making use of actual Rust idioms when transpiling C code into Rust, Citrus is choosing the opposite route. Instead of aiming to produce syntactically correct code that ignores Rust’s safety guarantees and idioms, as long as this will result in a Rust program that will be runnable, Citrus tries to find the closest approximation to idiomatic Rust reflecting a given C program, without expecting the outcome to run and not even giving guarantees that it will compile. Before using Citrus the C code needs to be heavily adapted in order to become transpilable, which one can see on their GitLab Readme.md [17] file under *Preparing C code for conversion*.

Just like with Corrode, we find that Citrus is no longer maintained. They even declare on

---

their GitLab page that "This project has been superseded by <https://C2Rust.com/>", which points to C2Rust's GitHub page.

*Citrus* is a tool initially taking a different and unique route, demarcating itself from the other major transpilation tools by being focused on approximating idiomacy rather than syntactical correctness, but was finally abandoned and superseded by *C2Rust*.

### 3.1.1.3 C2Rust

With Corrode abandoned and Citrus superseded by C2Rust, this tool is the only one left to pay attention to. C2Rust uses a C++ frontend for Clang and a Rust backend, both being actively maintained on GitHub.

From my experience in transpiling some programs with C2Rust, testing the C2Rust demonstration [11] on non-complex code and looking at an example input function from a C buffer library, presented by Per Larsen himself at the RustConf 2018, the possibly expected result of an initial C2Rust transpilation is non-idiomatic and needs a high amount of adaptation. In general such an initial transpilation can be `unsafe`, can use `no_mangle`, might call `malloc`, can have `C error handling`, `pointer access`, `casts` and, according to Per Larsen, it is likely that there will never be support for some language features (he mentions `longjmp`, `setjmp` and jumps in and out of GNU C statement expressions).

At the RustConf 2018 Per Larsen also talks about other limitations that existed by that time. Those concern `variadic function definitions` (Rust RFCs blocking issue #2137), `bitfields` (Rust RFCs blocking issue #314), `long double` and `_Complex` types (Rust libc blocking issue #355) as well as `macros`.

For now C2Rust's developers target the weakness of an initial C2Rust transpilation by suggestion a workflow that starts with a call to the C2Rust core program and continues with a procedure of rewriting, refactoring and cross-checking the result, partway using additional tools (they offer themselves) and partially involving manual effort. We will take a closer look at this workflow in the next subsection.

*C2Rust* is the only tool as of writing this thesis that is being actively maintained and it is also the most mature automatic C to Rust transpiler out there. Its bare application however suffers from weaknesses the developers openly address and for which they suggest a rather complicated workflow, which requires manual effort and familiarization with additional tools.

### 3.1.1.4 C2Rust Workflow

In this section we examine the workflow that is presented and suggested by *C2Rust*'s developers themselves more closely.

The workflow consists of stages that shall represent a current state of a program, initially being inherently *unsafe C* at *Stage 0*, and ideally ending at a stage, in which the programmer has produced idiomatic Rust that shows no divergence to the original C code's functionality, according to *C2Rust*'s *cross-checking components*' [10] evaluation.

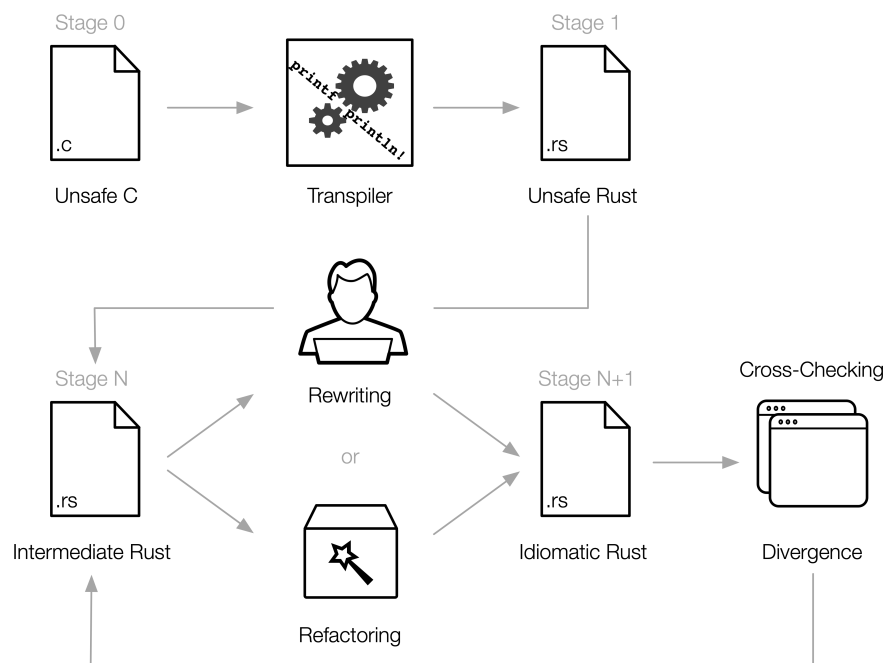


Figure 3.1: C2Rust Overview [9]

We will ourselves subdivide this workflow into *Steps*, which shall aid illustration of the concept.

---

### Step A

Consuming the unsafe C input, a transpilation with C2Rust is performed, where *Stage 1* represents this initial call's result. At this stage we expect the unidiomatic and unsafe Rust code that was produced in our own attempts of a bare C2Rust execution step. Confirming our previous statement, the developers expect the outcome at this stage to be unsafe themselves, entitling it *Unsafe Rust*.

### Step B

The programmer has to perform manual changes to the code in the stage denoted *Rewriting*.

### Step C

Manual rewriting can either be performed in a way that directly results in *Idiomatic Rust*, letting the programmer advance to *Stage N+1*, or can produce *Intermediate Rust*, where he or she will take the interim step to *Stage N*.

*Stage N* allows for refactoring to be performed by C2Rust's *Refactoring Tool* [13] in an attempt to let it produce idiomatic Rust automatically.

### Step D

The process leads to a stage in which the previously mentioned *cross-checking* tools are used to decide whether the newly created Rust diverges from the initial C. If there is no divergence, the programmer completed the workflow and should have successfully translated the given C code into functionally equivalent and (satisfyingly) idiomatic Rust; If not, he or she needs to return to **Step C**.

#### 3.1.1.5 Assessment of C2Rust's Workflow

C2Rust's developers present a workflow, which provides the programmer with a concept that can ideally lead to idiomatic Rust that is functionally equivalent to a given initial C input. However, there are issues with this workflow that suggest the assumption that its application is not easily realizable on larger projects.

First of all, it is left entirely open which amount of manual work is sufficient to lead to the point of making usage of C2Rust's refactoring components feasible. From my personal experience, the situation after an initial C2Rust transpiler call does not differ much from

---

---

the previous state, such that it seems to require a comparable amount of knowledge of the project structure and interiority to begin pacing towards a transformation into idiomatic Rust. Offering the possibility of generating idiomatic Rust from an intermediate state sounds persuasive, but it is hard to estimate how much prearrangement the code needs to undergo in order to be satisfactorily prepared. We need to remember that manual rewriting always is a prerequisite for entering the *Refactoring stage*, as shown in C2Rust Workflow.

Even when acquiring the requirements for C2Rust's refactoring tools, the process of calling them is not guaranteed to result in success. A *cross-checker* can be used to verify functional equivalence of initial C and momentary Rust. While this may seem like an optional opportunity in general, C2Rust's refactoring tools' output certainly needs verification. Either way, the programmer will be involved in additional effort. Either because he or she rather chooses to perform verification him- or herself (e.g. by writing unit tests), or because he or she needs to gain insight into the usage prerequisites and mechanisms of yet another tool.

Should the cross-checker find divergence at the final stage, the programmer is suggested to recede to the *Intermediate Rust* stage. This implies presence of a loop that is technically not guaranteed to terminate. This implication alone allows for the proposition that there is no general way to estimate whether manual C to Rust translation is less efficient than C2Rust's workflow for an input transcending a certain size or complexity. Its usefulness will have to be experimentally estimated in the following proceeding.

### 3.1.2 Hybrid Projects

In Automating Tools we already saw that automatic transpilation of C to Rust is still in an early stage in general, and even the most mature transpiler out there, *C2Rust*, has to overcome issues retaining it from being easy to use and broadly applicable on large and complex projects.

Besides the need for its complicated workflow to solve existing issues, another challenging aspect of its general approach has to be discussed. We already mentioned a couple of issues, arising when using C2Rust for a complete project transpilation; But also module extraction and isolation is difficult when it comes to large and complex projects, since most of them inherit long dependency chains, which in the end often means the need to transpile the main portion of the project for a unit of code to work (further deliberations in The Silver Searcher - Introduction and Data).

---

In codecare's Motivation for this Work codecare states that

[...] legacy system renewal is hardly ever economically viable via a "big bang" approach in which the legacy system is completely replaced all at once.

Since we elaborated that a "big bang" approach is exactly what is hard to circumnavigate using C2Rust, its economical value as a transpilation solution in real-world application seems nominal. As an alternative approach codecare further states that they

[...] especially consider those solutions which allow partial software modernisation in crucial locations in a localised manner

which confirms the importance of finding a way to transpile projects incrementally, without the need for complete knowledge of the respective project's code, and with some guarantee to allow for manual intervening when it comes to the length of respective calling chains subject to transpilation.

In this section we will take a look at projects that already successfully realize a similar approach of C to Rust transpilation, and try to learn from their knowledge and experience.

### 3.1.2.1 Mozilla Firefox

Mozilla Firefox is a very sophisticated project, which is open-source and is already having a remarkable amount of parts in Rust and running.

To analyze Mozilla's transpilation approach, the first step was to clone and build Firefox, following their build instructions for developers [28]. A count of Rust- and C-specific files shows that indeed a larger portion of the code is already transpiled into Rust.

```
find . -name "*.rs" | wc
```

shows 7497 hits,

```
find . -name "*.c" | wc
```

results in 4107 hits.

We do get 15692 hits though when utilizing *find* on *.h* files.

To clarify how they manage to keep their browser compilable and executable one needs to answer the following questions

- 
- Who is the main caller, C or Rust?
  - How is cross-language inter-operability managed inside this project?
  - Can we derive some sort of *recipe*<sup>4</sup> from Mozillas approach?

I decided to forward these questions to Mozilla's research department. While waiting for an answer, I assumed it would have been suspenseful to see to which extend these questions are answerable from the perspective of someone uninvolved. Had they replied, I would have been able compare their assertions with those private findings in an unbiased way.

When trying to analyze their code, I found that the project is large and taking away a generalizable approach of their C code substitution or Rust code integration methods would probably take weeks.

In the meantime I wrote another email to Bert Peers, who is mainly contributing to Mozilla by working on *Core components*<sup>5</sup>, and attracted my attention pushing Rust related commits [3] into the Firefox code - and he quickly replied. As an introduction statement he anticipates that he himself is [still] learning their systems as well and there was

a lot to grok for sure

so the former estimation of this project being too large for this context, seems appropriate.

Obviating the need for rapidly getting familiar with this large project's interiors, I am very thankful for his information, explanations and descriptions, that provided valuable insight into their strategy.

In his mail response he formulates the essence of what he knows about their management of C and Rust in the following way:

In short, I'm not sure that we have a real step where C code gets transpiled into Rust directly; rather we have two separate codebases, and with careful use of compatible data types, extern "C" interfaces, and some auto-generated glue, the two can be compiled separately and then all linked together into a single *xul.dll*.

---

<sup>4</sup>We will use this term in this thesis to describe our collection of transpilation steps that we will formalize; Thereby we will split the term into *build recipe* and *programming context recipe*.

<sup>5</sup>Shared components used by Firefox and other Mozilla software, including handling of Web content; Gecko, HTML, CSS, layout, DOM, scripts, images, networking, etc [61].



---

Bert Peers kindly invited me to the *gfx-firefox* chat group on *Matrix*, where I could ask a couple of questions to people actively working on porting graphics-related parts of Mozilla Firefox to Rust.

One active member, Jeff Muizelaar, Sr. Staff Software Engineer at Mozilla, approved Bert Peers' statement that there is no direct transpilation done in the browser code, but rather modular manual rewriting and linking. He pointed me to many other projects in which C or C++ code is transferred into Rust - the projects *librsvg*, *rustybuzz*, *raqote* and *rexpats* belonging to these.

I am extremely thankful for Peers' and Muizelaar's help and orientation in identifying and finding real world examples to gain substance for a theoretical creation of a recipe for such an incremental transition.

### 3.1.2.2 librsvg

In the *gfx-firefox* chat group Jeff Muizelaar pointed me to a couple of projects in which a transfer from C into Rust is practiced or already executed. One of these being *librsvg*, which he

[...] would hold [...] up as the prototypical example of an incremental transition of C code into Rust [32]

which emerged this candidate as one to examine closely.

On their Github page [44] one can quite easily identify the initial step in moving any part of this project from C to Rust [45].

Additionally, their commit naming-convention makes module-wise transpilation attempts clearly identifiable. For instance, in case of their *marker* module, which is an internal drawing machinery, the commit message says "Fully implement markers in Rust. rsvg-marker.c is gone!" [46].

The commit message "Start a branch to port bits of librsvg to Rust", rang in the transpilation of the *marker* module on the 25th of October 2016.

The commit issuing finalization of the marker module port was pushed on the 27th of February 2017, so there were about four months of work involved to fully transpile this module into Rust.

---

Having the dates from creation to finalization narrowed down precisely, I decided to replay this very module transition, such that I reset their project to the state before the initial Rust commit and filtered out only the minimal necessity of related commits.

A detailed description of this proposal can be found in Replaying librsvg's rsvg-marker Port.

### 3.1.2.3 rustybuzz

*rustibuzz* [75] is a project that is manually and incrementally converting *harfbuzz* [35], a text shaping library [36], from C++ to Rust [32]. There are some limitations as for their GitHub Readme, which are the reasons why we do not examine the project closer here.

This is mine yet another attempt to port *harfbuzz* to Rust. [...] The problem is that *harfbuzz* code is very interconnected and it's hard to swap out random parts. Also, despite having almost 1800 tests, there are still a lot of white spots which I will try to fill first. Since it's way too easy to miss some important edge-case.

### 3.1.2.4 rust-lzo

*rust-lzo* [75] is automatic conversion, performed by Jeff Muizelaar, using Corrode, which was abandoned before attempts to make it idiomatic [32]. For we are particularly interested in manual, incremental transpilation, we will not have a closer look in this context.

### 3.1.2.5 raqote

*raqote* [63] is another of Jeff Muizelaar's projects, in which he translates *full-scene-rasterizer* [29] into Rust, which is a "full-scene deferred rasterizer [...] [which] does  $4 \times 4$  super-sampling but is much faster than drawing the whole scene and then scaling down in final pass" [30].

Since this is a complete project translation, we will not examine any further here. We are mainly interested in C and Rust coexistence.

---

### 3.1.2.6 rexpats

*rexpats* [64] is a "[...] work-in-progress conversion from unsafe Rust, transpiled directly from *libexpats* [using C2Rust], into safe, idiomatic Rust code." [65], with *libexpats* [42] being a C library for parsing XML [43].

Its current state seems unstable or unfinished as their GitHub page says: "Do not use this in production (yet!), but help refactoring and rewriting is always welcome."

---

## 4 Testing Automation with C2Rust

---

As described in State of the Art Research some existent tools provide opportunities to automatically transpile C code into Rust. Among those C2Rust is most mature one, as well as the only one still actively maintained. In this chapter we will demonstrate usage of C2Rust during an attempt to automatically transpile *lz4*. We will thereby examine whether the impression of C2Rust's limitations of applicability in a context of larger projects upholds, or whether it enables us to easily produce functionally equivalent Rust.

---

### 4.1 lz4 - Introduction and Data

---

*lz4* [51] is a "lossless compression algorithm, providing compression speed  $> 500 \text{ MB/s}$  per core, scalable with multi-cores CPU [sic]. It features an extremely fast decoder, with [...] multiple  $\text{GB/s}$  per core, typically reaching RAM speed limits on multi-core systems." [52].

#### 4.1.1 Introduction to loc

"loc is a tool for counting lines of code. It's a rust implementation of cloc, but it's more than 100x faster." [49]. Its output is provided for each repository that gets examined closer in this thesis.

---

### 4.1.2 lz4: loc

Language	Files	Lines	Blank	Comment	Code
C	37	18063	2501	2371	13191
C/C++ Header	14	3425	519	1510	1396
Markdown	18	1881	492	0	1389
Makefile	10	1690	243	300	1147
HTML	2	976	161	0	815
Python	5	883	121	67	695
Plain Text	2	255	33	0	222
YAML	2	172	7	0	165
C++	1	248	40	48	160
Bourne Shell	5	159	26	12	121
Autoconf	3	76	4	7	65
Total	99	27828	4147	4315	19366

### 4.1.3 lz4: Repository Size

Size of cloned repository = 7.6 MB

Repository size after building = 8.9 MB

### 4.1.4 lz4: Tests

Comprehensive test cases exist and testing can be invoked via

```
make test
```

Its execution fails though for *dev* branch on *Arch Linux* at the 2nd of October 2020.

```
diff -s tmphf1 tmphf2
Files tmphf1 and tmphf2 are identical
make[1]: Leaving directory '.../lz4/tests'
make: *** [Makefile:127: test] Error 2
```

---

## 4.2 lz4 and C2Rust

---

In previous attempts to perform a full project transpilation with C2Rust, I recognized that there is a certain threshold of complexity to the initial C program that is hard to deal with in this context. One would struggle at having a single successful run of C2Rust, mainly due to language constructs that cannot be transpiled and formatting issues. The first would require a lot of C code preparation, in a sense of manual rewriting, the latter would bring the need of setting up very specific beautifier configurations, for instance with *uncrustify* or *ClangFormat*. Although this certainly is a possibility, the time it might take needs to stand in relation to the time effort of a manual translation to Rust. That being said, lz4 is an appropriate candidate for an automated transpilation assessment, as it is neither very complex nor hard to survey, but by no means trivial.

### 4.2.1 lz4: Transpilation with C2Rust

To illustrate a transpilation with C2Rust, I followed the steps described in their manual [12]. Before starting the actual transpilation process, they state the necessity to create a *compile\_commands.json* file inheriting all the compile commands used to build the original project that one chose as the target for transpilation. To automatically create this file, one can use *bear* [8] precedent to *make* in the original project's root folder (:= PROJECT).

```
cd PROJECT
bear make
```

After identifying lz4's main method being in *programs/lz4cli.c* we successfully transpiled the project, with *SOME\_CLANG\_PATH* and *SOME\_PATH* being variable to the respective location on the executing machine:

```
CLANG_PATH=SOME_CLANG_PATH C2Rust transpile -b lz4cli
↪ SOME_PATH/lz4/compile_commands.json
↪ -output-dir=SOME_PATH/lz4_rust
```

In the newly transpiled project main folder, we denoted as *lz4\_rust* in our transpilation command, we did a

```
cargo build
```

---

which will give us some warnings, but no errors. This is not generally guaranteed; In many cases the code must be heavily prepared for a C2Rust run to complete without errors.

In our transpilation command we provided `-b lz4cli` where `-b` means that we want a binary to be produced by C2Rust. This binary was now executable via

```
SOME_PATH/lz4cli -c SOME_FILE.SOME_EXTENSION SOME_FILE.lz4
```

where we needed to prevent accidentally calling a potentially system-provided lz4 version by explicitly pointing to our newly created one, lying in `SOME_PATH`; while `-c` sets `stdout` as our default output.

## 4.2.2 lz4: Solving Cargo Build Warnings

This seemed like a suiting first step. I wanted to start off with a clean project and thereby gain some knowledge about present Rust compiler complaints, before getting my hands on manual rewriting or refactoring.

These are the types of warnings introduced as part of transpiling:

- (1) warning: unnecessary parentheses around method argument
- (2) warning: unreachable statement
- (3) warning: unused variable
- (4) warning: unused comparison that must be used
- (5) warning: unused logical operation that must be used
- (6) warning: unused import

### warning (1)

*unnecessary parentheses around method argument*

Looking at all instances of this warning, I found that these parentheses all were unnecessary indeed, so they could simply be removed. This is a shortened snippet from the actual code, where we can see the double parentheses before and after the *if-else-statement*.

```
nextSrcSizeHint = (*dctx).tmpInTarget.wrapping_add((if
↳ (...) {...} else {...})).wrapping_add(BHSize);
```

---

## warning (2)

### *unreachable statement*

There were two instances of this warning. At the respective equivalent C passages the code is commented, saying although the return statements were indeed unreachable, some compilers complained about their absence.

```
...
        /* fallthrough */
        return XXH64_avalanche(h64)
    }
    /* impossible to reach */
    __assert_fail(
        b"0\x00" as *const u8 as *const libc::c_char,
        b"xxhash.c\x00" as *const u8 as *const libc::c_char,
        806 as libc::c_int as libc::c_uint,
        (::std::mem::transmute::<&[u8; 76],
        &[libc::c_char; 76]>(b"U64 XXH64_finalize(U64,
        ↪ const void *, size_t, XXH_endianness,
        ↪ XXH_alignment)\x00")).as_ptr());
    return 0 as libc::c_int as U64;
    /* unreachable, but some compilers complain without it */
```

This probably does not count for Cargo and I decided that we can pretty safely remove these code blocks.

The formatting and structure of this code snippet also visualizes the difficulties the programmer will find him- or herself in, when aiming at more idiomacy. It is very unreadable and confusing and there is certainly a lot of effort involved in figuring out what each of the code blocks does and whether a feasible, adequate transformation of the original C code has been applied by C2Rust.

## warning (3)

### *unused variable*



---

This warning was solvable pretty straight forward, as cargo already told us to prefix unused variables with an underscore, which would silence those warnings.

```
warning: unused variable: `s`
--> src/lib/lz4.rs:1772:9
   |
1772 |     let s: *mut libc::c_void =
   |         ^ help: consider prefixing with an underscore: `_s`
   |
= note: `#[warn(unused_variables)]` on by default
```

Deciding whether or not such a variable could be removed altogether was too early at this stage, as I did not have a funded understanding of the code, yet.

#### warning (4)

*unused comparison that must be used*

This warning appeared two times. Here we needed to further analyze the code in order to get an idea about the issue. On the warning's first occurrence the initial C code is performing an assertion<sup>1</sup>.

```
if (tableType==byPtr) assert(dictDirective==noDict);
```

In the transpiled Rust code there was no such thing performed; Although Rust's `assert`<sup>2</sup> is a language construct that exists in its standard library. Here the transpilation produced a boolean expression, the outcome of which does not have any subsequent consequence:

---

<sup>1</sup>This is a macro that implements a runtime assertion, which can be used to verify assumptions made by the program and print a diagnostic message if this assumption is false. When executed, if the expression is false (that is, compares equal to 0), `assert()` will write information about the call that failed on `stderr` and then call `abort()`. [7]

<sup>2</sup>Asserts that a boolean expression is true at runtime. This will invoke the `panic!` macro if the provided expression cannot be evaluated to true at runtime. [6]

```
(tableType as libc::c_uint) == byPtr as libc::c_int as  
↳ libc::c_uint;
```

This is the *if* condition from the original code, without an *if* and with no body. Additionally the cast to *libc::c\_int* can be omitted when casting to *libc::c\_uint* afterwards anyway. The line below indicates that another assertion is missing from the original code, where it read

```
assert(acceleration >= 1);
```

which was already pointing towards the strong possibility that C2Rust is somehow not translating assertions at all; but this time we could just solve it, while already at it.

The new Rust code is:

```
if (tableType as libc::c_uint) == byPtr as libc::c_uint {  
    assert_eq!(dictDirective, noDict)  
}  
  
assert!(acceleration >= 1);
```

The second occurrence of this warning was solved halfway analogously. Again there was a boolean expression with no *if* and no body, and another assertion that was missing entirely. Additionally, many constant values defined in the C code are missing in this Rust transpilation, as we could get from the context. It seemed very questionable already where this would lead to, but there was still hope that we might have come to a point at which C2Rust's cross-checker (see C2Rust Workflow) could possibly do some magic. I added the *if* and the assertion to the body, but was not able to construct the other assertion without changing much code and needed to keep this in mind for later.

## warning (5)

*unused logical operation that must be used*

This warning was caused by another conditional expression without its condition or body. The context is to support *sparse-file mode*, which will require further analysis of the whole

---

code, in order to find out why this is needed in the original C, and whether it is still needed here.

```
(!f.is_null()) && (*prefs).sparseFileSupport != 0;
```

### warning (6)

*unused import*

This warning is hard to understand and probably misleading. It tells us that

```
use ::lz4_rust::*;
```

was an unused import, but removing it, will result in an error that is hard to trace back, saying "error: Linking with 'cc' failed!". We need to add back this import and silence the warning by putting

```
#[allow(unused_imports)]
```

above it.

This is strange behavior that I did not come across in any manual transpilation. It is self-contradicting, as either the import is needed and the warning is wrong, or it is not, but then there should be no error occurring from its removal. To the best of my knowledge I can only interpret this as another sign of the transpilation being broken.

We are for now warning-free, but came across many code passages that already showed that this cannot be a clean functionally equivalent transpilation, and that one will probably not circuit the need of complete knowledge of the original C code, if one was to continue this route and start off with an attempt to now debug the new Rust.

### 4.2.3 lz4: Assessment of Manual Effort for Functional Equivalency

In the previous sections we could already see that the transpiled code cannot be functionally equivalent to the original C. It is questionable whether the effort needed to reverse-debug the outcome of a complete transpilation with *C2Rust* can generally reduce manual transpilation effort in larger projects.

---

*lz4*, with 15553 lines of code and a size of 7.6 MB (see *lz4*: loc), is not a huge project, but we can see that the code transpiled by C2Rust is very hard to understand and is basically broken in several ways. Imagining how difficult code analysis of larger and more complex projects would be after a C2Rust transpilation, makes it very clear, that this cannot be the optimal strategy to approach a C to Rust transition.

*C2Rust*'s developers suggest a workflow which requires a cyclic alternation of automatic and manual work. This however seems to be very complex and brings diverse other issues and uncertainties, which let me maneuver away from the automatic approach.

Due to all these circumstances we will rather look at different manual and *incremental* approaches, in which C and Rust can coexist, while Rust can slowly and steadily replace more and more C code, like this is already successfully done in some projects (see Hybrid Projects).

---

## 5 Rust Background Information

---

In this chapter we will introduce basic Rust concepts that we used and applied for definition of our transpilation `recipe` and actual implementations.

---

### 5.1 Rust Concepts

---

#### 5.1.1 Mutability

"For a place expression<sup>1</sup> to be assigned to, mutably borrowed, implicitly mutably borrowed, or bound to a pattern containing `ref mut` it must be mutable. We call these mutable place expressions. In contrast, other place expressions are called immutable place expressions." [90]

In particular, all Rust variables are `immutable` by default, meaning one cannot reassign them a value.

This would **not** work:

```
let x = 0;  
x = 5; // error here
```

---

<sup>1</sup>"A place expression is an expression that represents a memory location. These expressions are paths which refer to local variables, static variables, dereferences (`*expr`), array indexing expressions (`expr[expr]`), field references (`expr.f`) and parenthesized place expressions. All other expressions are value expressions." [92]

---

while this would:

```
let mut x = 0;  
x = 5;
```

## 5.1.2 Ownership

As opposed to other programming languages, where memory needs to be manually freed, and such that make use of a `garbage collector`<sup>2</sup>, Rust manages memory "[...] through a system of ownership with a set of rules that the compiler checks at compile time. None of the ownership features slow down your program while it's running" [91].

In Rust, each value has a variable that is its `owner`. There cannot be multiple owners of a piece of data. In Rust a memory location cannot be pointed to more than once at a time.

### 5.1.2.1 References and Borrowing

References allow to refer to some value without taking ownership of it. Just as variables are immutable by default, so are references. One can have only one mutable reference to a particular piece of data in a particular `scope`. At any given time, one can have either one mutable reference or any number of immutable references. Having references as function parameters is called `borrowing`.

[93]

## 5.1.3 Lifetimes

In Rust, every reference "has a lifetime, which is the scope for which that reference is valid. Most of the time, lifetimes are implicit and inferred, just like most of the time, types are inferred. We must annotate types when multiple types are possible. In a similar way, we

---

<sup>2</sup>"In computer science, garbage collection (GC) is a form of automatic memory management. The garbage collector, or just collector, attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program." [98]

---

must annotate lifetimes when the lifetimes of references could be related in a few different ways. Rust requires us to annotate the relationships using generic lifetime parameters to ensure the actual references used at runtime will definitely be valid". During our transpilation we never had to deal with `lifetime` annotations, so we will not examine any further at this point.

#### 5.1.3.1 The Borrow Checker

The Rust compiler has a borrow checker that compares scopes to determine whether all borrows are valid. At compile time, Rust compares the size of the two lifetimes and decides whether an inner lifetime block is smaller than its outer one. In this case the program would get rejected, otherwise accepted.

[94]

---

## 5.2 Rust Organization

---

### 5.2.1 Cargo

"Cargo is the Rust package manager. Cargo downloads your Rust package's dependencies, compiles your packages, makes distributable packages, and uploads them to crates.io [84], the Rust community's package registry" [83].

---

## 5.2.2 Modules

```
Syntax:  
Module :  
    mod IDENTIFIER ;  
  | mod IDENTIFIER {  
    InnerAttribute*  
    Item*  
  }
```

Figure 5.1: The Rust Reference - Module

In Rust, a "module is a container for zero or more items. A module item is a module, surrounded in braces, named, and prefixed with the keyword `mod`. A module item introduces a new, named module into the tree of modules making up a `crate`" (see Crates). "Modules can nest arbitrarily."

[89]

## 5.2.3 Crates

```
Syntax  
Crate :  
    UTF8BOM?  
    SHEBANG?  
    InnerAttribute*  
    Item*
```

Figure 5.2: The Rust Reference - Crate



---

In Rust, a "crate is a unit of compilation and linking, as well as versioning, distribution, and runtime loading. A crate contains a tree of nested module scopes. The top level of this tree is a module that is anonymous (from the point of view of paths within the module) and any item within a crate has a canonical module path denoting its location within the crate's module tree."

[86]

#### 5.2.4 Extern Crate Declarations

**Syntax:**

*ExternCrate* :

```
extern crate CrateRef AsClause? ;
```

*CrateRef* :

```
IDENTIFIER | self
```

*AsClause* :

```
as ( IDENTIFIER | _ )
```

Figure 5.3: The Rust Reference - Extern Crate

In Rust, an "extern crate declaration specifies a dependency on an external crate. The external crate is then bound into the declaring scope as the identifier provided in the extern crate declaration. The as clause can be used to bind the imported crate to a different name."

[87]

#### 5.2.5 Functions

In Rust, a "function consists of a block, along with a name and a set of parameters. Other than a name, all these are optional."

---

We must know that we can return from a function in C manner.

```
fn multiply(a: i32, b: i32) -> i32 {  
    return a * b;  
}
```

But there is another way to return from a Rust function. We need to consider that the "block of a function is conceptually wrapped in a block that binds the argument patterns and then returns the value of the function's block. This means that the tail expression of the block, if evaluated, ends up being returned to the caller.". This is why we can also rewrite the above like this (note that the semicolon is also omitted):

```
fn multiply(a: i32, b: i32) -> i32 {  
    a * b  
}
```

It must be emphasized that this way of returning is only viable for *the tail expression of the block*, so the following would **not** work:

```
fn max(a: i32, b: i32) -> i32 {  
    if a > b { a }  
    else { b }  
}
```

Instead we could write (note that the semicolon still can respectively be omitted):

---

```
fn max(a: i32, b: i32) -> i32 {  
    if a > b { return a }  
  
    b  
}
```

[88]

---

## 5.3 C and Rust Interoperability Types

---

A basic overview of C/C++ types and their most likely analogous Rust types can be found in *locka99.gitbooks.io* [50]. Rust's `libc` [73] and `cty` [72] crates provide **interoperability types**, creating **aliases** for the analogous Rust types. These allow for C and Rust communication.

---

## 6 Approaching a Module Porting Recipe

---

In this chapter, we will go through the process of acquiring a methodical approach to perform module porting from C to Rust.

First, there will be a discussion about issues and compromises one has to consider in the regard of defining what can be seen as a `module` in this context. Then we will be replaying *librsvg's rsvg-marker port* in order to crystallize out the essence of their successful porting procedure.

---

### 6.1 Identifying Modules

---

The first instinct is better not to wildly start translating the first C or header file one comes across. One probably wants to make sure that translation will lead to a point at which some portion of the code will be entirely encapsulated in Rust, without the need to further translate other portions. Otherwise one would feel like basically having to translate the whole project for anything to work. Ultimately, we want to find a way to *incrementally* transpile from C to Rust, which implies this sort of boundary from one part of the code to the rest - being independently enclosed. But, during the further progress, this turned out to be manageable, and even true, only to a certain extend. In fact it is hard to even imagine a project in which one part of the code exists quietly next to the rest. Most likely one will find some units of meaning, like a server or a client, but there will have to be some sort of communication between them, immediately meaning that one will find a C caller to our Rust.

This turned out to be a very important insight, as it strongly affects the way in which we can write our Rust, in order to function with the rest of the project, still being in C.

The issue arising from this is clearly how to deal with C input in our Rust code. Most considerably, will will often find C pointers to C structs or arrays, which seems to bring us the same issues we wanted to avoid in the first place, namely dealing with memory

---

---

allocation. This fortunately is not what it comes down to in the end. Rust already offers some types and methodical approaches that address this very issue and once we created this crossover we can benefit from Rust's automatic memory management again. This would usually mean unsafe type conversion, but this is most likely as good as it gets. Especially when starting to translate one part of the code that is called by C and has to hand its own output to some C code again. I believe that these are compromises one has to accept when starting a project translation in an incremental fashion, but that there is this long-term benefit that exposes itself more strongly the more Rust is present inside the project; Meaning that eventually a point will get achieved at which enough Rust is present to drop C approaches entirely and rewrite present code to be significantly more idiomatic.

---

## 6.2 Replaying librsvg's rsvg-marker Port

---

### 6.2.1 librsvg - Introduction and Data

librsvg is "A small library to render Scalable Vector Graphics (SVG), associated with the GNOME Project. It renders SVG files to Cairo surfaces. Cairo is the 2D, antialiased drawing library that GNOME uses to draw things to the screen or to generate output for printing.". It can render non-animated SVGs to a Cairo surface with a minimal API. [44]

---

### 6.2.1.1 librsvg: loc

Language	Files	Lines	Blank	Comment	Code
Rust	105	36773	5259	3667	27847
C	15	4067	801	266	3000
Markdown	6	1445	375	0	1070
Makefile	9	955	152	134	669
Bourne Shell	2	690	77	161	452
C/C++ Header	6	2034	166	1513	355
XML	3	351	38	0	313
Autoconf	7	285	40	25	220
Plain Text	5	236	23	0	213
Python	3	273	41	40	192
Toml	5	163	20	3	140
Batch	1	42	10	0	32
YAML	1	23	3	0	20
CSS	3	46	7	22	17
Total	171	47383	7012	5831	34540

### 6.2.1.2 librsvg: Repository Size

Size of cloned repository = 52 MB

Repository size after building = 1,6 GB

### 6.2.1.3 librsvg: Tests

323 test cases exist, of which 44 fail for *master* branch on *Arch Linux* at the 3rd of October 2020. Testing can be invoked via

```
make distcheck DESTDIR=/tmp/foo
```

and results in:

```
# TOTAL: 323
# PASS: 278
# SKIP: 0
# XFAIL: 0
```

---

```
# FAIL: 44
# XPASS: 0
# ERROR: 1
```

### 6.2.2 Approaching marker Port Replay

With Jeff Muizelaar declaring this project the prototypical, incremental C to Rust transition [32] and successful identification of the last C-only project state, as well as the end of rsvg-marker port from C to Rust, being done, I decided trying to create a fork of libsvg. I hard-reset this project clone back to the last C-only state. Subsequently, I looked at changes to rsvg-marker only, trying to create some commits reflecting the steps they took to get there, so I could extract their recipe and learn from it.

The difficulty was getting to the state of successful rsvg-marker porting while ignoring as much as possible of all other changes to this large project that had happened in the meantime. Having a minimal version of one module port only would allow me to get the essence of necessities for such a conversion, and ideally maybe even something close to a formula that I could learn, reuse and generalize.

### 6.2.3 Filtering Commits

My first step for replaying the rsvg-marker port was narrowing down the timeframe of the module transpilation. This was easily done due to their clear commit comments (see libsvg). Once I knew the beginning of Rust integration and module port finalization I was able to use GitHub's search function to look for Rust and marker related commits, while only focusing on results within the ascertained span.

These I split into two groups first. One of which can be taken as is, identifying commits that are related to marker functionality in a contentual manner, and one that potentially exposes structural insights.

Among the latter, we can expect certain places of interest. These being:

The Rust specific *Cargo.toml* that "is about describing your dependencies in a broad sense, and is written by [the programmer]" [15];

Rust's *lib.rs* crate<sup>1</sup> that contains the program's logic and *main.rs* for potential command line parsing logic [77];

as well as some files for build instructions, like configuration files or a *Makefile*.

---

<sup>1</sup>A a compilation unit in Rust; Crate is compiled to binary or library. [67]

---

## 6.2.4 Gathering Necessary Rust Files

In a first commit to my librsvg fork, I gathered all Rust specific files, without needing to look at their interiors. These are the previously introduced *lib.rs* and *Cargo.toml*; A *Cargo.lock*, which "contains exact information about your dependencies [,] is maintained by Cargo and should not be manually edited" [15], as well as the new *marker.rs* file and all its dependencies.

Tracing down dependencies for Rust files can be done in a similar fashion to C projects. While in a C project one would follow *headers* (\*.h files), here one would examine *use* declarations, found in form of \*.rs files.

## 6.2.5 rsvg-marker.c Removal and rsvg-marker.h Adjustment

In this commit I deleted the rsvg-marker C version, as our new Rust one is already present in our project. The rsvg-marker.h file's content has shrunk down, but interestingly the file itself still exists. This has to be done this way, when we want part of our Rust to be callable by the outside C code.

## 6.2.6 Adjusting Makefile.am

librsvg is using a *Makefile.am*, which is consumed by *automake* to create a *Makefile.in*, which finally is used by a configuration script (here it is called *configure.ac*) to build the Makefile. This is the part inside the project where we can see how the coexistence of C and Rust is managed in librsvg. The main idea is to let *gcc* manage the C code compilation as before, while *Cargo* is taking over the Rust part. In a next step they are both linked together, in this case by invoking *ldd*.

This is achieved by first listing all relevant Rust files in the *Makefile.am*, including *Cargo.toml* and all \*.rs files. In *librsvg* this list is called *RUST\_SOURCES*. The *Cargo.lock* file is the only entry in the newly created *RUST\_EXTRA* list.

Finally, a target directory for a static library is defined.

### librsvg: Static Library Definition

```
TARGET_DIR=@abs_top_builddir@/target/@RUST_TARGET_SUBDIR@
RUST_LIB=$(TARGET_DIR)/librsvg_internals.a
```



---

They specify that all previously listed Rust files will be the content of the static library to create.

#### librsvg: Static Library Creation

```
$(RUST_LIB): $(RUST_SOURCES)
  cd $(top_srcdir)/rust && \
  CARGO_TARGET_DIR=@abs_top_builddir@/target cargo build
```

To leave the rest to the linker, they pass the newly created static library *RUST\_LIB* to *ldd*, in this specific case to *rsvg\_view\_3\_LDADD*. In addition *RUST\_SOURCES* and *RUST\_EXTRA* are distributed by adding them to *EXTRA\_DIST*.

### 6.2.7 Enabling Cross-Compiling

This might be very project specific. In case of librsvg they are using an *AM\_CONDITIONAL* to check whether cross-compilation is enabled in *configure.ac*.

#### librsvg: Cross-Compiling

```
AM_CONDITIONAL(CROSS_COMPILING, test $cross_compiling =
↪ yes)
```

### 6.2.8 Optional Check for Cargo and rustc

To check whether Cargo and rustc are installed on the machine that the project is being compiled on, librsvg uses *AC\_CHECK\_PROG*.

---

#### librsvg: Check for Cargo and Rust

```
AC_CHECK_PROG(CARGO, [cargo], [yes], [no])
AS_IF(test x$CARGO = xno, AC_MSG_ERROR([cargo is required.
↳ Please install the Rust toolchain from
↳ https://www.rust-lang.org/]))

AC_CHECK_PROG(RUSTC, [rustc], [yes], [no])
AS_IF(test x$RUSTC = xno, AC_MSG_ERROR([rustc is required.
↳ Please install the Rust toolchain from
↳ https://www.rust-lang.org/]))
```

### 6.2.9 Optional Debug / Release Switch

librsvg allows to specify an `-enable-debug` command line option, to create a development build. By default the project is built in public release mode.

### libsvg: Debug Release Switch

```
AC_ARG_ENABLE(debug,  
  AC_HELP_STRING([--enable-debug],  
    [Build Rust code with debugging information  
    ↪ [default=no]]),  
  [debug_release=$enableval],  
  [debug_release=no])  
  
AC_MSG_CHECKING(whether to build Rust code with debugging  
  ↪ information)  
if test "x|\$|debug_release" = "xyes" ; then  
  AC_MSG_RESULT(yes)  
  RUST_TARGET_SUBDIR=debug  
else  
  AC_MSG_RESULT(no)  
  RUST_TARGET_SUBDIR=release  
fi  
AM_CONDITIONAL([DEBUG_RELEASE], [test "x|\$|debug_release"  
  ↪ = "xyes"])  
  
AC_SUBST([RUST_TARGET_SUBDIR])
```

---

## 7 Formalization of librsvg's Build Recipe

---

In this chapter, we will formalize *librsvg's* approach of transitioning from C to Rust from the build part perspective, in order to achieve more abstraction in the sense that we decouple this approach from their project, while on the other hand we concretize execution steps, such that we can apply the result on foreign projects.

We already talked about the basics of setting up cross-compilation for C and Rust, using *gcc* and *Cargo* in Formalization of librsvg's Build Recipe. This is not everything one has to consider when incrementally moving from C to Rust, but it quickly enables the programmer to start off with the fundamentals.

In Replaying librsvg's rsvg-marker Port we saw concrete code snippets used in *librsvg's* *Makefile.am* and *configure.ac* files, which can basically be directly adapted if you find yourself in a project that uses a similar build concept. Here I will assume a project setup quite similar to librsvg's and present their approach in a step by step way. I will talk about a more generalized way to handle certain aspects of instructions that are suggested here in upcoming chapters. This shall be an illustration of how dual compilation is managed in this specific, but not uncommon, case.

- Inside the project root create a *rust* folder and change to this new directory.
- Invoke *cargo init --lib* to create a library setup, including a *src/lib.rs* and *Cargo.toml*, but no *main.rs* file, which we will not need for this approach.

---

### Cargo init --lib: Directory Structure

```
project-root/  
└─ rust/  
    └─ src/  
        └─ lib.rs  
        └─ Cargo.toml
```

- Specify a name for the static library to be created (without the "lib" prefix) in *Cargo.toml*

### Cargo.toml: Static Library Declaration

```
[lib]  
name = "rust_internals"  
crate-type = ["staticlib"]
```

- In the *Makefile.am* specify *RUST\_SOURCES*. We can start off by adding *Cargo.toml* and *lib.rs* and add the files we are about to write in Rust later on.

### Makefile.am: RUST\_SOURCES

```
RUST_SOURCES = \  
    rust/Cargo.toml \  
    rust/src/lib.rs
```

- Define *RUST\_EXTRA* for *Cargo.lock*.  
This file will be auto-created by *Cargo* during the first compilation attempt.

---

#### Makefile.am: RUST\_EXTRA

```
RUST_EXTRA = \  
    rust/Cargo.lock
```

- Define how the static library file (which will be named "librust\_internals" due to our declaration in *Cargo.toml*) will be created.

#### Makefile.am: Static Library Creation

```
TARGET=@abs_top_builddir@/target/@RUST_TARGET_SUBDIR@  
RUST_LIB=$(TARGET)/librust_internals.a  
  
$(RUST_LIB): $(RUST_SOURCES)  
    cd $(top_srcdir)/rust && \  
    CARGO_TARGET_DIR=@abs_top_builddir@/target cargo  
    ↪ build
```

- Add *RUST\_LIB* to *ldd*. The respective flag needs to be identified in the specific project. Typically it will be called something like *<project\_name>\_LDD*.

#### Makefile.am: Static Library to ldd

```
<project_name>_LDD += $(RUST_LIB)
```

- *RUST\_SOURCES* and *RUST\_EXTRA* can be distributed across the project by adding them to the most often already present *EXTRA\_DIST*.

---

### Makefile.am: Static Library to ldd

```
EXTRA_DIST += \  
  $(RUST_SOURCES) \  
  $(RUST_EXTRA)
```

- Finally, to make your life easier it is probably best to define some test and clean commands for our Rust.

### Makefile.am: Static Library to ldd

```
check-rust:  
  cd $(srcdir)/rust && \  
  CARGO_TARGET_DIR=@abs_top_builddir@/target cargo  
  ↪ test  
  
clean-rust:  
  cd $(top_srcdir)/rust && \  
  CARGO_TARGET_DIR=@abs_top_builddir@/target cargo  
  ↪ clean
```

The way we set things up, the build process can simply be triggered by invoking *make* in the usual fashion.

We could now also supplement the optional additions from the librsvg rsvg-marker port replaying, if everything works like expected.

---

## 8 Module Porting to Rust

---

In this chapter we will work with our build recipe, previously defined in Formalization of librsvg’s Build Recipe. After a simple introductory example, in which we will apply the fundamental technical basics for C and Rust coexistence, we will take a look at our incremental transpilation of *calc*, which is a bit more complex and offers the possibility to get to know Rust language features needed for conversion and C and Rust interoperability. Finally, examination of our larger partial project transpilation of *The Silver Searcher* will allow us to solve issues in a real-world environment and dealing with some additional Rust features and transpilation challenges.

*librsvg*’s attempt to both incrementally and manually transfer their code into Rust, seemed right away like an opportunity to define a methodical approach, which could be easily applied on foreign projects. Trying to limit the noise of non-related commits, in combination with the idea of creating an own fork of their project, turned out to generate a good overview to the principles of their C and Rust coexistence.

The following will illustrate the application of module porting in a non-redundant way, meaning that each section will build upon knowledge of the previous one. Also, only unseen language features will get introduced and discussed; and their complexity is meant to increase as we proceed.

---

### 8.1 A Simple Example

---

To give an impression on how a minimal project consisting of C and Rust could look like, we can imagine a simple adder module, which does nothing but add two numbers and return their sum, written in Rust, and a C caller, that wants to print this result to the standard output.



---

### 8.1.1 C Implementation

We can quickly create a C program fulfilling its purpose of simply converting two numbers passed as command line parameters into integers and printing out their sum, by calling a foreign function *add* from a foreign file *adder.h*.

#### A Simple Example: adder.h

```
1  #ifndef ADDER_H
2  #define ADDER_H
3
4  int add(int a, int b);
5
6  #endif
```

#### A Simple Example: adder.c

```
1  #include "adder.h"
2
3  int add(int a, int b)
4  {
5      return a + b;
6  }
```

### A Simple Example: main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "adder.h"
5
6  int main(int argc, char* argv[])
7  {
8      int a = atoi(argv[1]);
9      int b = atoi(argv[2]);
10
11     printf("The sum of %d and %d is %d.\n", a, b, add(a, b));
12
13     return 0;
14 }
```

For convenience we also supply a Makefile.

### A Simple Example: Makefile

```
1  TARGET := add
2  C_SOURCES := $(shell ls *.c)
3  CC := gcc
4
5  default: all
6
7  all:
8      $(CC) $(C_SOURCES) -o $(TARGET)
9
10 clean:
11     rm -f $(TARGET) *.o
```

---

## 8.1.2 Porting Adder to Rust

When only wanting to port the adder module to Rust and leave the rest of the program in C, the simplicity of our C program already allows us to picture some of the typical C to Rust transpilation issues.

C can only know about foreign files, when their declaration is found in a header file, but Rust does not work with header files. It natively does not split function declaration and definition across two files, like this is usually done within well-structured C programs; In Rust there is only the \*.rs file for both.

Additionally, we can only provide C types as arguments to our Rust functions, whereby we know that this cannot work out of the box with arbitrary C types.

The Embedded Rust Book fortunately offers suggestions on how to maneuver out of these issues, by showing how to integrate Rust in a C-compatible way [2].

The first fact to realize is that Rust has no stable ABI<sup>1</sup> (meaning "you have to compile and link everything all in one go on the same version of the Rust compiler" [70]), which is why they suggest the C ABI for any interoperability between C and Rust.

Then, there is name mangling, which C and Rust are doing differently. Due to the previous settlement to rather make Rust C compatible than the other way around, we have to change Rust's name mangling for all functions we want to expose to the outside C world. This is done via the `#[no_mangle]` function attribute.

Furthermore, we need to make sure that our Rust function only uses C-compatible types. For the sake of simplicity, we are only dealing with integers in this case, for which we can use the type `cty::c_int`. We will need to add `cty` under *dependencies* in *Cargo.toml*.

### A Simple Example: Cargo.toml Dependencies

```
[dependencies]
cty = "0.2.1"
```

---

<sup>1</sup>In computer software, an application binary interface (ABI) is an interface between two binary program modules; often, one of these modules is a library or operating system facility, and the other is a program that is being run by a user.[4]

---

The latest dependency version number can always be obtained from *crates.io* [72].

To finally build a bridge between our Rust function and our C code, we can in fact use our C header file unchanged. The Rust function needs to explicitly be public to be callable from C, which is achieved by the `pub` keyword; It additionally needs to provide information about being a C header's implementation, which requires proving *extern "C"* to the *function declaration*. The function's name must not deviate from the C header naming, which also applies to all input parameters.

#### A Simple Example: `adder.rs`

```
#[no_mangle]
pub extern "C" function add(a: cty::c_int, b: cty::c_int)
    -> cty::c_int {
    a + b
}
```

Now that the implementation details are sorted out, we can start applying our build recipe.

Creating a *rust* directory at the top level, applying the *cargo init -lib* command inside this directory, and moving our *adder.rs* file to *rust/src*, results in the following file structure, with *simple\_adder* being our project's name.

#### A Simple Example: Directory Structure

```
simple_adder/
├── adder.h
├── main.c
├── Makefile
├── rust/
│   ├── src/
│   │   ├── adder.rs
│   │   └── lib.rs
│   └── Cargo.toml
```

Since part of our approach is building a static archive from the Rust and then linking it

---

with all present C code, we need to expose our previously created `adder.rs` file as a module in *lib.rs*. This is as easy as specifying the file name without its ending.

#### A Simple Example: `lib.rs`

```
mod adder;
```

For the creation of your static Rust library, we just need to declare it in *Cargo.toml*, which will now be complete.

#### A Simple Example: `Cargo.toml`

```
[package]
name = "rust"
version = "0.1.0"
authors = ["Philip Koslowski <philip.koslowski@gmail.com>"]
edition = "2020"

[dependencies]
cty = "0.2.1"

[lib]
name = "rustadder"
crate-type = ["staticlib"]
```

As a final step we adapt our Makefile to build the static library and link it.

### A Simple Example: Makefile C & Rust

```
TARGET := add
C_SOURCES := $(shell ls *.c)
CC := gcc
C_FLAGS := -lpthread -ldl
RUST_LIB := rust/target/debug/librustadder.a

default: all

all:
    cd rust && cargo build
    $(CC) $(C_FLAGS) $(C_SOURCES) $(RUST_LIB) -o $(TARGET)

clean:
    rm -f $(TARGET) *.o
```

Two common errors we would receive here are

```
undefined reference to `pthread_attr_getstack`
undefined reference to `dlsym`
```

which we can solve by providing the respective C\_FLAG.

```
C_FLAGS := -lpthread -ldl
```

In the end we must not forget to delete adder.c before we run our program.

```
rm adder.c
make clean && make
./add 2 4
The sum of 2 and 4 is 6.
```

---

## 8.2 calc - Introduction and Data

---

calc [14] is "a simple command-line based calculator" by *Vivek Kannan* [96]. In comparison to our previous example though, it is complex enough to show some deeper insight into how we can deal with more versatile C input. In the chapter A Simple Example we only dealt with integers, while here we can have a look at handling constant values, static data and C pointers as input parameters to our Rust.

### 8.2.1 calc: Project Structure

#### calc: Project Structure

```
calc
├── .gitignore
├── LICENSE
├── README.md
├── calc.c
├── helpers.c
├── helpers.h
└── struct.h
```

#### 8.2.1.1 calc: loc

Language	Files	Lines	Blank	Comment	Code
C	2	493	129	0	364
Markdown	1	108	44	0	64
C/C++ Header	2	29	7	0	22
Total	5	630	180	0	450

#### 8.2.1.2 calc: Repository Size

Size of cloned repository: 176 KB

Repository size after building: 204 KB

---

### 8.2.1.3 calc: Tests

No tests exist for this project.

## 8.2.2 Dealing with C Pointers in Rust

Just like with integers, where we can use `cty::c_int` as a C compatible Rust type, we can use `*const cty::c_char` as a compatible type for a `const char[]` and `*mut cty::c_char` for a pointer to a non-constant C array. Instead of `cty` we can choose `libc`, which also offers these types.

`*const` and `*mut` are raw pointers without safety or liveness guarantees. Derefencing such a raw pointer is *unsafe*, but can be used to convert it into a reference by re-borrowing it (`&*` or `& mut*`). Their use is generally discouraged, but they do exist for the very purpose to support interoperability with foreign code (and for writing performance-critical or low-level functions). [62]

In many cases we have no choice other than using them, since our C caller will frequently pass over a C array. The best we can do is accept them as a function argument to communicate with our C, but then convert them into a non-discouraged Rust type, like a `vector`, we can further work with.

Again, the more Rust will get introduced in our project, the more likely it will become that we might drop their usage entirely, but for now we follow this incremental approach, in which we cannot work around these interoperability types.

From my experience, the situation in which a C array is handed over as a parameter to a function we want to use with our new Rust, occurs very frequently. This is why it makes sense for me to define some functions right away, which will convert these arrays into a Rust type, we would work with in an idiomatic environment.

Within this project, we can immediately see, that we will need to deal with a lot of C `char arrays`, which we would rather represent as a `String`, we could pass as a reference, or directly as a `&str` in Rust.

Here are two functions I use very frequently to change them into the other back and forth (we will also very often have to return a C compatible type).



---

### String to Char Pointer and Back

```
fn str_to_c_char_ptr(s: &str) -> *mut libc::c_char {  
    let c_str = CString::new(s.as_bytes()).unwrap_or_default();  
    return c_str.into_raw() as *mut libc::c_char;  
}  
  
unsafe fn char_ptr_to_string(s: *const libc::c_char) -> String {  
    return String::from(CStr::from_ptr(s).to_str().unwrap());  
}
```

We can see that these conversions can be really tricky and one certainly does not want to reinvent the wheel in each of these occasions. This is why defining these functions provides a lot of convenience, and more clarity to the code.

In our first function *str\_to\_c\_char\_ptr* we transferred our *&str* into a *CString*, which is "a type representing an owned, C-compatible, nul-terminated string with no nul bytes in the middle. This type serves the purpose of being able to safely generate a C-compatible string from a Rust byte slice or vector." [81].

Since *s* is neither, we first had to convert it into its bytes-representation by using *as\_bytes()*. Creating a new *CString* returns a *Result*, which "is [an enum] that represents either success (*Ok*) or failure (*Err*)" [26].

Since we want to get the value behind the *Result* type, we need to call *unwrap\_or\_default()*. We could also use *unwrap()* directly, but the documentation says: "Because this function may panic, its use is generally discouraged. Instead, prefer to use pattern matching and handle the *Err* case explicitly, or call *unwrap\_or*, *unwrap\_or\_else*, or *unwrap\_or\_default*." [59]. I personally think that this is really up to the programmer and situation and I will not be consistent with this myself.

Due to its C-compatibility, the *CString* could now be converted into a raw pointer using *into\_raw()* and be cast into a *\*mut libc::c\_char*, which we wanted to return.

In our second function *char\_ptr\_to\_string* we created a *CStr* from the pointer first, by calling *from\_ptr*. Just like intuition would tell us, "&*CStr* is to *CString* as &*str* is to *String*: the former in each pair are borrowed references; the latter are owned strings." [80]. Analogously to the above, we get a *Result* value by calling *to\_str()* and obtain the value by calling either of the *unwrap* variants. We return an owned *String* via *String::from*, which has the advantage of not having to worry about the lifetime of a borrowed reference.

---

### 8.2.3 Dealing with Constant Values and Static Data in Rust

Constant values in Rust are pretty straight-forward. Whenever we would write `#define` for defining constant values in C, we can go with `pub const` in Rust, where all we have to add is an explicit type declaration.

#### Constant Value in C

```
#define PI 3.14159265358979323846;
```

#### Constant Value in Rust

```
pub const PI: libc::c_double = 3.14159265358979323846;
```

We need to import our constant value though, where we want to use it.

#### Importing a Constant Value

```
use crate::crate_name::PI;
```

With `crate_name` being the file name without the `.rs` ending the constant value is declared with.

Whenever we wanted to declare statics that require heap allocation, like a `vector`, we ran into a problem. We can declare a *public constant vector* or a *public static vector*, but we can not initialize it before program execution. If we try to create a vector in the following way

#### Incorrect Initialization of a Static Vector

```
pub static vector_name: Vec<i32> = vec![1, 2, 3];
```

we will get an error message saying: "calls in static are limited to constant functions, tuple structs and tuple variants" and the analogous error message for will also appear using `const`.

---

The other theoretical way, pushing elements to a declared but uninitialized vector, requires a runtime function call, so this will not work either.

There is a crate solving this problem, called *lazy\_static*. "Using this macro, it is possible to have statics that require code to be executed at runtime in order to be initialized. This includes anything requiring heap allocations, like vectors or hash maps, as well as anything that requires function calls to be computed." [53]. So this allows us exactly what we were not allowed before.

#### Correct Initialization of a Static Vector in Rust

```
use lazy_static::lazy_static;

lazy_static! {
    pub static ref vector_name: Vec<i32> = {
        let mut v: Vec<i32> = Vec::new();
        v.push(1);
        v.push(2);
        v.push(3);
        v
    };
}
```

In our calc project, I replaced the *helpers* module with Rust. In the C version of this module, there is a *char\** to operators that are used and a *char\** array to functions that can be passed as *char\**.

#### helpers.c: Public Initialized Data

```
1 char* operators = "+-*/%^$~_!";
2 char* functions[] = { "sin", "cos", "tan", "asin", "acos",
    ↪ "atan", "sinh", "cosh", "tanh", "asinh", "acosh", "atanh",
    ↪ "exp", "floor", "ceil", "round", "log", "ln", "sqrt", "abs",
    ↪ "sgn" };
```

The program logic requires to get the position of an element within this data, which is

---

done by a matching during a loop over each element inside.

#### helpers.c: Finding Element's Position in the Array

```
1  #define FUNC_COUNT 21
2
3  int isFunction(char* s) {
4      int i = FUNC_COUNT - 1;
5      for(; i >= 0 && strcmp(functions[i], s) != 0; i--);
6      return i;
7  }
```

In our Rust version, we could use an array as a data type and iterate over each element in the same fashion; But due to our previous insight, we rather used a static `HashMap`, which can store the respective position as the element's key, which would diminish the need of iteration. Thereby, we were also enabled to deal with a potential error resulting from an input that is non-existent in the given array, by introducing the *None* case in *isFunction*.

### helpers.rs: HashMap Replacing Array

```
lazy_static! {
    pub static ref functions: HashMap<String, libc::c_int> = {
        let mut map = HashMap::new();
        map.insert(String::from("sin"), 0);
        map.insert(String::from("cos"), 1);
        map.insert(String::from("tan"), 2);
        map.insert(String::from("asin"), 3);
        map.insert(String::from("acos"), 4);
        map.insert(String::from("atan"), 5);
        map.insert(String::from("sinh"), 6);
        map.insert(String::from("cosh"), 7);
        map.insert(String::from("tanh"), 8);
        map.insert(String::from("asinh"), 9);
        map.insert(String::from("acosh"), 10);
        map.insert(String::from("atanh"), 11);
        map.insert(String::from("exp"), 12);
        map.insert(String::from("floor"), 13);
        map.insert(String::from("ceil"), 14);
        map.insert(String::from("round"), 15);
        map.insert(String::from("log"), 16);
        map.insert(String::from("ln"), 17);
        map.insert(String::from("sqrt"), 18);
        map.insert(String::from("abs"), 19);
        map.insert(String::from("sgn"), 20);
        map
    };
}

#[no_mangle]
pub extern "C" fn isFunction(s: *mut libc::c_char) ->
    ↪ libc::c_int {
    let s_str = unsafe { char_ptr_to_string(s) };
    match functions.get(&s_str) {
        Some(position) => *position,
        None => -1
    }
}
```

---

## 8.2.4 Dealing with Multiple if-Statements in Rust

In our calc project's `helpers.c` file, we find a function `isSymbol` that uses many if-statements, calling `strcmp` over and over.

### helpers.c: Multiple if-Statements

```
1 double isSymbol(char* s) {
2
3     if(strcmp(s, "e") == 0)
4         return E;
5
6     if(strcmp(s, "pi") == 0)
7         return PI;
8
9     if(strcmp(s, "inf") == 0)
10        return INFINITY;
11
12    if(strcmp(s, "rand") == 0) {
13        srand((unsigned) time(NULL) + rand());
14
15        return (double) rand() / (double) RAND_MAX;
16    }
17
18    if(strcmp(s, "g") == 0)
19        return G;
20
21    return 0.0;
22 }
```

In Rust there is a convenient way of doing this, using `match`.

---

### helpers.rs: *match* Replacing Multiple if-statements

```
use rand::Rng;

#[no_mangle]
pub extern "C" fn isSymbol(s: *mut libc::c_char) ->
↳ libc::c_double {
    let s_str = unsafe { char_ptr_to_string(s) };
    match s_str.as_str() {
        "e" => E,
        "pi" => PI,
        "rand" => rand::thread_rng().gen:::<libc::c_double>(),
        "inf" => INF,
        "g" => G,
        _ => 0.0
    }
}
```

---

## 8.3 The Silver Searcher - Introduction and Data

---

In The Silver Searcher's GitHub Readme.md, this program is described as "a code searching tool similar to ack, with a focus on speed" [95].

In comparison to the previous introductory example program and *calc*, this project is big and complex. It is quite difficult to understand the code right away and it will take time to obtain an overview. Therefore it is well-suited for demonstrating how to deal with Rust integration without having to understand large portions of the code before being able to begin. I will also show how we can automate some parts of the process of integration, using *rust-bindgen* [71], which is less error-prone than the manual way of creating the needed interfaces and thereby can also save time.

---

### 8.3.0.1 The Silver Searcher: loc

Language	Files	Lines	Blank	Comment	Code
C	12	4634	491	211	3932
C/C++ Header	11	1464	167	144	1153
Bourne Shell	8	447	55	40	352
Markdown	3	505	182	0	323
Autoconf	1	75	17	0	58
Makefile	2	68	16	1	51
Python	1	29	8	3	18
Plain Text	1	2	0	0	2
Total	39	7224	936	399	5889

### 8.3.0.2 The Silver Searcher: Repository Size

Size of cloned repository = 3.4 MB

Repository size after building = 5.2 MB

### 8.3.0.3 The Silver Searcher: Tests

43 tests exist for this project and all of them pass at the 3rd of October 2020 on Arch Linux. They can be invoked via

```
make test
```

## 8.3.1 An Overview of rust-bindgen

When finding oneself in a larger C project and wanting to translate parts of into Rust, a good start is to get familiar with *rust-bindgen*, which will also be referred to as *bindgen* inside the documentation [1].

When starting to translate C code, chances are that one will find oneself in a situation in which one chases down function calling chains; meaning function A calls function B, which calls function C and D and so on. This can quickly get out of hand and summon up the very circumstance the programmer actually might have wanted to avoid in the first place - namely basically having to translate the whole project, if just blindly staying on



---

this route.

Taking a simple project like *calc* can mean easy module extraction and one might not have to worry about issues like these, but the larger and the more complex the project gets, the sooner it will become clear that an incremental approach can only work, if somehow the programmer was able to still call some C functions without having to transform them into Rust just yet.

To see what is meant, consider the dependency graphs generated by the documentation tool *Doxygen* [25] for *The Silver Searcher's* *search.h* and *search.c* files.

When looking at the files that directly or indirectly include *search.h*, it looks very easy to obtain an overview about this module.

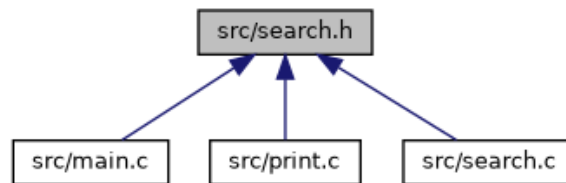


Figure 8.1: The Silver Searcher: Files Including *search.h*

While when we take a look at *search.h's* and *search.c's* dependency graph, it gets clear that this module is not encapsulated quite this easily.

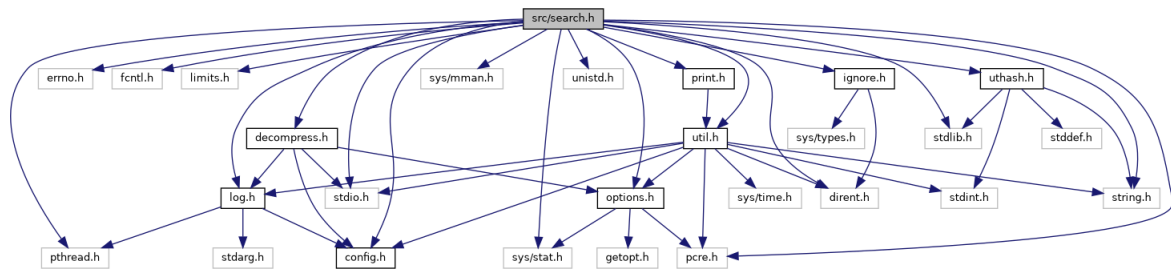


Figure 8.2: The Silver Searcher: search.h Dependency Graph

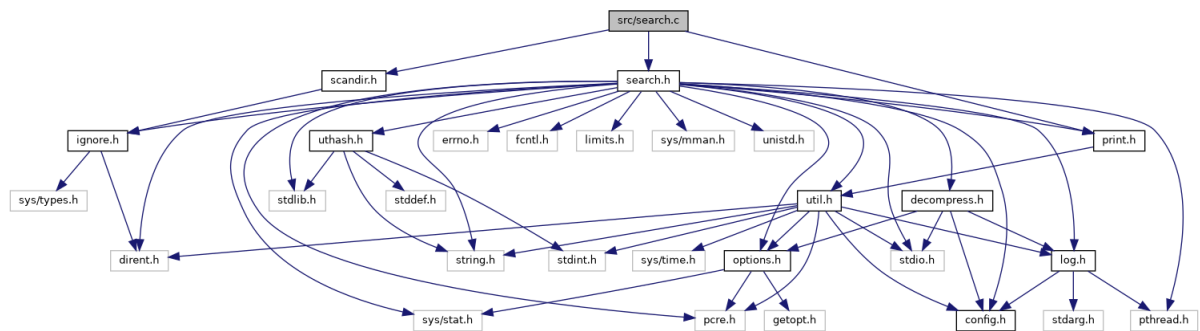


Figure 8.3: The Silver Searcher: search.c Dependency Graph

When arriving at a point where the programmer wants to leave parts of the translation for later, or live with the compromise of calling some C function in his new Rust code, he or she can do so, by creating a Rust compatible interface of these C constructs, which will most often be *functions* or *structs*.

If we wanted our Rust to access a C function declared in the following way:

#### **some\_function in C**

```
int some_function(const char* a, int b);
```

we would need to create an interface as follows.

#### **some\_function in Rust**

```
extern "C" {  
    pub fn some_function(a: *const cty::c_char, b: cty::c_int) ->  
        ↪ cty::c_int;  
}
```

For *structs* this could look like this.

#### **some\_struct in C**

```
struct some_struct {  
    char **a;  
    size_t b;  
};  
typedef struct some_struct some_struct;
```

#### **some\_struct in Rust**

```
#[repr(C)]  
#[derive(Debug, Copy, Clone)]  
pub struct some_struct {  
    pub a: *mut *mut cty::c_char,  
    pub b: size_t  
}
```

We see two `attributes` we did not come across, yet, but typically want to use whenever we are planning to access a C struct inside our Rust.

---

`#[repr(C)]` is a `data layout strategy` having a fairly simple intent: "do what C does. The order, size, and alignment of fields is exactly what you would expect from C or C++." [5]. Not using this attribute would mean that we were not able to access C data, so we actually need this whenever using C structs within Rust.

The `#[derive]` attribute provides basic implementations for derivable `traits` referred to in its parameter list.

`Debug` allows to format a value using the `{:?}` formatter;

`Clone` creates `T` from `&T` via a copy;

And `Copy` gives a type 'copy semantics' instead of 'move semantics'.

[24]

While the programmer is free in his choice of doing this himself, *rust-bindgen* provides a way to automate the whole process, auto-generating these interfaces, which can be very convenient in a big project.

### 8.3.2 Using rust-bindgen

After applying our build recipe steps, which I already demonstrated in Formalization of libsvg's Build Recipe, we can start setting up everything for automatic creation of our bindings.

First we need to gather all C header files we want to use functions from and put them into a single header file we call *wrapper.h*.

For this paper I translated *The Silver Searcher's filter* and *ignore* functionalities in *ignore.c*, replacing its main content in *ignore.rs*, so I already know which C functions and structs I would eventually use. Otherwise, we could collect them as we proceed.

#### ignore.rs: Includes from Bindings

```
use crate::bindings::{
    dirent, scandir_baton_t, ignores,
    log_debug, pcre_exec,
    opts,
};
```

---

They are declared in *options.h*, *util.h* and *scandir.h*, so my *wrapper.h* file would include these. We need to point to them relatively from the directory the *wrapper.h* file is created in. The important parts of our directory structure at this point looks like this.

#### The Silver Searcher: Directory Structure - wrapper.h

```
the_silver_searcher/  
├── src/  
│   ├── Makefile.am  
│   └── rust/  
│       ├── src/  
│       │   ├── lib.rs  
│       │   ├── Cargo.toml  
│       │   └── wrapper.h
```

This is our *wrapper.h* content.

#### The Silver Searcher: wrapper.h

```
#include "../src/options.h"  
#include "../src/util.h"  
#include "../src/scandir.h"
```

The next step is to create a script file called *build.rs*. Placing this file "in the root of a package will cause Cargo to compile that script and execute it just before building the package." [69].

### The Silver Searcher: Directory Structure - wrapper.h

```
the_silver_searcher/  
├── src/  
│   ├── Makefile.am  
│   └── rust/  
│       ├── src/  
│       │   ├── lib.rs  
│       │   └── build.rs  
│       ├── Cargo.toml  
│       └── wrapper.h
```

We fill it with the following content.

### The Silver Searcher: build.rs

```
1  extern crate bindgen;  
2  
3  fn main() {  
4      println!("cargo:rerun-if-changed=wrapper.h");  
5  
6      let bindings = bindgen::Builder::default()  
7          .ctypes_prefix("cty")  
8          .header("wrapper.h")  
9          .rustfmt_bindings(true)  
10         .generate()  
11         .expect("Unable to generate bindings");  
12  
13     bindings  
14         .write_to_file("src/bindings.rs")  
15         .expect("Couldn't write bindings!");  
16 }
```

In line 1 we point *Cargo* to an `extern crate bindgen`, which provides us the main functionality for the automatic interface generation. This crate has to be provided in the usual manner, looking for the latest version from *crates.io* and putting it under *Cargo.toml*'s dependency section.

---

In the main function, line 4, we tell *Cargo* to re-run the whole script if *wrapper.h* changes.

Lines 6 to 13 define our *bindings*. First we tell *bindgen* to build these with default values, and then we add instructions line after line.

*ctypes\_prefix("cty")* enables *cty* compatibility;

*header("wrapper.h")* points to the file where we generate our bindings from;

*rustfmt\_bindings(true)* tells to format the bindings file we are about to create

and finally *generate()* will produce it.

If something should go wrong we propagate an error message via *expect*, saying "*Unable to generate bindings*".

In lines 13 to 15 we tell the script where to place our file and how to name it via *write\_to\_file*. Again we let the script produce an error message in case of failure by adding *expect* with a custom error message.

Now, the file *bindings.rs* should automatically appear when compiling the project, leaving us with this final directory structure.

#### The Silver Searcher: Directory Structure - *bindings.rs*

```
the_silver_searcher/  
├── src/  
│   ├── Makefile.am  
│   └── rust/  
│       ├── src/  
│       │   ├── bindings.rs  
│       │   └── lib.rs  
│       ├── build.rs  
│       ├── Cargo.toml  
│       └── wrapper.h
```

We can now simply declare *bindings* as a *module* inside our *lib.rs*, so we can access it from within our Rust via *use crate::bindings::{...}*. We also add some attributes suppressing some typical warnings considering formatting we expect from bindings creation, particularly due to the merge of C and Rust coding conventions.

---

#### The Silver Searcher: lib.rs

```
#![allow(non_upper_case_globals)]
#![allow(non_camel_case_types)]
#![allow(non_snake_case)]
#![allow(unused)]

mod bindings;
```

### 8.3.3 The Silver Searcher - filename\_filter

The Silver Searcher is a command line tool that allows the user to visualize and localize occurrences of search queries inside of files, like one could do by using `ack` or `grep` in a Linux environment. The user can thereby add some options, allowing pattern matching, using regular expressions, as well as ignore patterns, case and (hidden) files. The `filename_filter` function with its subroutines is responsible for all of these, and the main part of the percentual program execution time is spent inside of it [19].

This is the part of the program we translated and present in the following.

#### 8.3.3.1 The Silver Searcher - filename\_filter: output

The Silver Searcher outputs the file name and line of occurrence inside a file in which a search query is found.

To provide a small example of how this search tool is used and what `filename_filter` does, let us look at a simple directory structure.

#### The Silver Searcher: Directory Structure - bindings.rs

```
test_dir
├─ someFile.test.txt
└─ someOtherFile.txt
```



---

*someFile.test.txt* and *someOtherFile.txt* both are containing only one line reading "test". Invoking `ag`, which is the name of *The Silver Searcher's* executable, searching for the word "test" inside of *test\_dir*

```
cd test_dir
ag test .
```

produces the following output

```
someFile.test.txt
1:test

someOtherFile.test.txt
1:test
```

which is what we expected.

We can now add a *.ignore* file to our *test\_dir* folder and add "\*.test.txt" to it, telling it to ignore all files with this ending while searching in the directory the *.ignore* file is located in.

#### The Silver Searcher: Directory Structure - bindings.rs

```
test_dir
├── .ignore
├── someFile.test.txt
└── someOtherFile.txt
```

Another call of

```
ag test
```

now produces

---

```
someOtherFile.txt
1:test
```

so we successfully ignored the query matching in *someFile.test.txt*, since we included its extension in *.ignore*, which is an example for one of many functionalities of the *filename\_filter* function.

### 8.3.4 Dealing with Iterators in Rust

One of *filename\_filter*'s subroutine tasks is returning a given file's extension.

#### Getting File Extension in C

```
const char *extension = strchr(filename, '.');
if (extension) {
    if (extension[1]) {
        // The dot is not the last character, extension starts at the
        ↪ next one
        ++extension;
    } else {
        // No extension
        extension = NULL;
    }
}
```

Rust already offers a method

```
pub fn extension(&self) -> Option<&OsStr>
```

via the *std::path::Path* crate, which behaves like this:

Extracts the extension of *self.file\_name*, if possible.

The extension is:

- None, if there is no file name;

- 
- None, if there is no embedded .;
  - None, if the file name begins with . and has no other .s within;
  - Otherwise, the portion of the file name after the final .

[55]

Unfortunately we cannot use this method here, since we want to reproduce only almost the same behavior, with the exception that we do not want the portion of the file name after the last dot, but rather after the first one (in case other criteria lets it qualify as an extension).

So, if we had defined a String *s* with the following content,

```
let s = "/data/folder1/text.test.example.txt";
```

calling

```
let ext = extension(&s);
```

would result in (the unwrapped) *ext* being *"txt"*, while what we wanted was *"test.example.txt"*.

We wrote our own method though, and thereby got to know iterators in Rust.

### helpers.rs: get\_extension

```
1 pub fn get_extension(filename: &str) -> Option<String> {
2     let mut v: Vec<char> = filename.chars().collect();
3     let len = v.len();
4     let pos = v.iter().position(|&c| c == '.');
5
6     if len < 2 ||
7        v[0] == '.' && v.iter().filter(|&c| *c == '.').count() < 2 ||
8        pos.is_none() {
9         return None
10    }
11
12    let mut s = String::new();
13    for i in pos.unwrap()+1..len {
14        s.push(v[i]);
15    }
16
17    Some(s)
18 }
```

The method `get_extension` takes one parameter `filename`, which is a `&str` and returns an `Option<String>`, meaning that the returned result can either be `None` or `Some`, while in the latter case a `String` can be unwrapped from the `Option` type.

We pass in an `&str`, being a non-iteratable type, which is why we have to convert it, which we do in line 2. We can easily produce a `Vector` from a `&str`, by first calling the `chars` method. Doing so creates a struct `std::str::Chars` that is "an iterator over the chars of a string slice" [79]. Then we call the `collect` method which "can take anything iterable, and turn it into a relevant collection" [54].

In the following we handle the cases in which we want our method to return `None` and those where we want to return `Some String`, the `String` itself being our extension.

In line 3 we simply store the number of elements in our `Vector` (that is the number of chars in our input) and save it for later in a variable `len`.

In line 4 we also store a variable (`pos`), holding the position of the first dot in our `Vector`. Here we simply created an iterator from a collection (our `Vector v`), using the `iter` method, which iterates over `&T` [57], and then called `position`, where "`position()` takes a closure

---

that returns *true* or *false*. It applies this closure to each element of the iterator, and if one of them returns *true*, then *position()* returns *Some(index)*. If all of them return *false*, it returns *None*." [58].

Closures in Rust are "anonymous functions you can save in a variable or pass as arguments to other functions." [18]. The argument in our *position* call

```
|&c| c == '.'
```

can be seen as an input parameter to a function circumscribed by the vertical bars and a condition that is applied on each element in the original collection, until it evaluates to *true* for the first time. The calling method decides how this result is eventually presented; in this case by returning the position in the collection of this very element.

Not all iterator methods using closures will return on the first time a condition is met. Some of them will for instance keep track of all matches within the collection; one of them being *filter*. Its call produces a struct that is an "iterator that filters the elements of *iter* with *predicate*." [82]. We use this method in line 7 again matching for dots in our collection. This time we are interested in the number of occurrences though, which is why we supply the *count* method on its result.

In lines 6 to 9 we define three conditions in an *or-relation* that can be read as follows:

The extension is *None*:

- If the file name has 0 or 1 characters;
- If the file name begins with . and has no other .s within;
- If there is no embedded .

In all other cases we conclude that we have some extension. We want to return an *Option<String>*, so we first define an empty String, *s*, in line 12. Now we can iterate over the collection starting from the first element after the first dot, pushing the *character* at each position onto the String, by using the *push* method in line 14.

Finally, in line 17, we return the String containing the extension, wrapped in *Some*.

---

### 8.3.5 Dealing with C's Fixed Sized Character Arrays in Rust

In *The Silver Searcher's* `helpers.rs` file, we defined a method that takes a fixed sized `char` array and returns a `String`.

#### helpers.rs: Fixed Length Char Array to String

```
pub fn fl_c_char_ptr_to_str(fixed_arr: &[cty::c_char; 256]) -> String {
    let ptr_arr = fixed_arr.as_ptr();
    assert!(!ptr_arr.is_null());

    unsafe { char_ptr_to_string(ptr_arr) }
}
```

Here we return our `char_ptr_to_string` method, defined in *Dealing with C Pointers in Rust*, after turning the array into a pointer by calling `as_ptr` [60] and making sure that we do not deal with a NULL input in the `assert!` call.

Here is a snippet from *filename\_filter* showing how we transformed the original C input into idiomatic Rust, we were able to further work with.

#### The Silver Searcher: filename\_filter snippet in C

```
const char *filename = dir->d_name;

...

    if (strncmp(filename, "./", 2) == 0) {
#ifdef HAVE_DIRENT_DNAMLEN
        filename_len = strlen(filename);
#endif
        filename++;
        filename_len--;
    }

...
```

### The Silver Searcher: filename\_filter snippet in Rust

```
let filename: &str = fl_c_char_ptr_to_str(&(*dir).d_name);
let mut filename_vec: Vec<char> = filename.chars().collect();

...

if filename_vec[0] == '.' && filename_vec[1] == '/' {
    filename_vec.remove(0);
}

...
```

### 8.3.6 Dealing with Double Pointers in Rust

We already saw how to deal with pointers in Rust and how to represent them in a way that can be used in idiomatic Rust.

Things can get a bit more tricky when dealing with double or triple pointers. Here is an example, used as a helper function in *The Silver Searcher's* filter and ignore functionality transpilation.

#### helpers.rs: double\_i8\_ptr\_to\_vec

```
1 pub unsafe fn double_i8_ptr_to_vec(ptr: *mut *mut i8, ptr_len:
  ↳ usize) -> Vec<String> {
2     let pv = slice_from_raw_parts(ptr, ptr_len);
3     let mut ptr_vec: Vec<String> = Vec::new();
4
5     for i in 0..ptr_len {
6         let elem = (&*pv)[i];
7         ptr_vec.push(char_ptr_to_string(elem));
8     }
9
10    ptr_vec
11 }
```

---

The input to *double\_i8\_ptr\_to\_vec* is a *\*mut \*mut i8*, where *i8* is the actual type behind *cty::c\_char*, while *ptr\_len* represents the number of elements of *ptr*.

We do not want to work with non-iteratable C-compatible types and avoid guaranteeing memory-safety ourselves. Our output shall therefore be a *Vec<String>*.

First we need to discard the issue that we cannot access single elements in our initial data representation. When trying to access an element, we would get the following error message.

```
cannot index into a value of type `*mut *mut i8` rustc(E0608)
```

Therefore we used a function

```
pub fn slice_from_raw_parts<T>(data: *const T, len: usize) -> *const [T]
```

[31]

from *std::ptr* to transform our *\*mut \*mut i8* into a *\*const [\*mut i8]*, meaning we got an *iteratable raw slice*. As we can see this functions needs to know the number of elements behind the pointer allocation, which is why we need to pass *ptr\_len* to *double\_i8\_ptr\_to\_vec* in the first place.

As we can read in the documentation, making this function call alone would not require our function to be denoted *unsafe*, but actually using the function's returned value would. Again, there is no other way to create a state in which C and Rust co-exist, than performing unsafe operations once in while; Alternatively we could not even aim at being any idiomatic.

After calling *slice\_from\_raw\_parts* and save the result as *pv* in line 2 we create a new *Vector<String>* in line 3.

From line 5 to 8 we loop over the provided *pointer length* where we now can access elements (in form of *cty::c\_char*) by indexing.

To do so, we have to dereference *pv*.

```
let elem = (*pv)[i];
```



---

Then we can convert the respective element, which is a *\*mut i8*, into a `String` by using our previously introduced helper function *char\_ptr\_to\_string*, after which we can simply push the element into our vector.

```
ptr_vec.push(char_ptr_to_string(elem));
```

After the iterations are performed, we return the result in line 10.

### 8.3.7 Dealing with C's NULL in Rust

In idiomatic Rust there is no `NULL` type. The closest counterpart are the `None` and `Some` types we already got to know (see *Dealing with Iterators in Rust*).

Nonetheless, there might be times when converting a C type into an idiomatic Rust type would mean the same hassle as just using it in its *interoperable* state, usually provided via `libc` or `cty`.

During the transpilation of *The Silver Searcher's* filter and ignore functionalities this situation occurred a couple of times, but there is already a chance that one can get a `NULL` input when the initial caller provides the C function.

To check whether something is `NULL` in Rust (where this "something" has to be an interoperability type), we can use *is\_null* from `core::ptr::const_ptr`.


```
pub fn is_null(self) -> bool
```

[56]

Since we are trying to get rid of these interoperability types as quickly as possible, we did not need to further deal with `NULL` during the whole transpilation process.

### 8.3.8 The Silver Searcher: Transpilation Results

After porting *filename\_filter* and its subroutines to Rust, the project is still compilable and executable. There is no need to change any compilation or execution command. The project will build fine invoking



---

```
./configure && make
```

just like before.

Calling

```
make test
```

will confirm success, as all 43 tests are still passing.

---

## 9 Related Work

---

In this chapter we will take a look at related work, dealing with Rust's safety promises and limits. The challenges to the high expectations to Rust's unique concepts cannot only be perceived on a high level; It turns out that `unsafe Rust` is not only exposed directly by the programmer, but may also be provided indirectly by the introduction of transitive dependencies within a given call chain. Here we will summarize attempts to identify these issues and provide ways to avoid their occurrence, as well as such that deal with them, rather than diminishing their presence.

---

### 9.1 Is Rust Used Safely by Software Developers?

---

In this section, I will summarize Ana Nora Evans', Bradford Campbell's and Mary Lou Soffa's paper about Unsafe Rust [27]. They come up with the question of whether Rust realistically can stand the impact of being that safe a language as it propagates itself to be. They show limits to safety guarantees within the language, transitive dependencies as an often overseen cause for divergence of perceived and factual safety, and they propose methods to improve Rust by offering ways on how to deal with some uncovered safety risks.

#### 9.1.1 Why Does Unsafe Rust exist?

There are several use cases that Rust's safety enforcement would make impossible to realize, if there was no Unsafe Rust. Among these are

- Configuring hardware
- Reading a network socket

- 
- Arbitrary memory accesses with C-style pointers
  - Invoking system calls
  - Calling foreign functions (usually C functions)
  - Executing inline assembly instructions
  - Eliding bounds checks for performance
  - Accessing global static memory

Unsafe Rust is embedded to allow access to a machine's hardware and to support low-level performance optimizations. It contains support for operations that are difficult to statically check, such as C-style pointer memory access and mutable global variables. Using such features, the compiler is unable to statically guarantee Rust's safety properties.

The paper is about performing a large-scale empirical study to explore how software developers are using Unsafe Rust in real-world Rust libraries and applications.

One can split Unsafe Rust into three categories.

#### *Unsafe Operations*

- Calling a function marked unsafe, non-Rust external function, or a compiler intrinsic (a function whose implementation is handled specially by the compiler)
- Dereferencing a C-style pointer.
- Accessing a mutable global variable.
- Using inline assembly instructions.
- Accessing a field of a union type.

#### *Unsafe Functions*

Some functions will be marked unsafe because of possibilities to access unallocated memory under certain conditions.

Conversely, a function containing unsafe operations may provide a safe interface.

#### *Unsafe Traits*

A trait may be declared unsafe if it contains any unsafe method or its implementations must satisfy an invariant.

---

### 9.1.2 What Does Unsafe Rust do?

Unsafe Rust disables some, but not all, of Rust’s compiler checks for certain regions of the code.

Programmers using Unsafe Rust are responsible for writing code free of safety violations and undefined behavior; however, what constitutes undefined behavior is currently not well specified and can change with different versions of the compiler. This situation makes safely using Unsafe Rust difficult.

### 9.1.3 What Is Analyzed in This Paper?

The authors of this paper

1. Determine how frequently the unsafe keyword is used
2. Analyze the call graph of every function in their data set to identify if at any point the function may use code that is not safe and not checked by the compiler

This analysis is supposed to enable the authors to find code that looks safe, but is actually Unsafe Rust.

Furthermore they

- Identify the underlying code behavior that necessitates the use of Unsafe Rust to analyze the frequency of the various unsafe operations
- Observe the use of Unsafe Rust over time to see whether there are evolving changes in the community

### 9.1.4 How Do the Authors Perform Their Analysis?

The authors developed and implemented an algorithm which handles generic polymorphism, constructing *Extended Cell Graphs*. The algorithm produces two cell graph versions (for functions with runtime polymorphism)

- A *conservative* version (assuming the call will be to unsafe code)

- 
- And an *optimistic* one (assuming the call will be to code statically checked to be safe)

They then traverse and analyze the resulting *extended call graph* to determine how unsafety propagates in real-world Rust code.

### 9.1.5 Analysis Results

Software engineers use the keyword `unsafe` in less than 30% of Rust libraries, but more than half cannot be entirely statically checked by the Rust compiler because of Unsafe Rust hidden somewhere in a library's call chain.

After analyzing over 85% of the valid Rust libraries available at the start of their study, they find that 29% contain at least one explicit use of `unsafe`.

When considering the dependency tree, however, that number increases to around 50%, meaning half of Rust libraries use Unsafe Rust or rely on other libraries using Unsafe Rust themselves.

#### 9.1.5.1 Popular Libraries

Narrowing down to just the most used and downloaded libraries increases the use of Unsafe Rust, as around 60% of popular crates include it.

#### 9.1.5.2 Majority of Unsafe Usage

The majority of unsafe uses in the Rust ecosystem are calls of other Rust functions that are marked `unsafe`. They find that only 22% of these unsafe functions are to external libraries implemented in C, suggesting that a majority of the Unsafe Rust is actually from Rust code where the software developer decided to disable the compiler checks.

Finally, they see negligible increases in the frequency of `unsafe` used over the past ten months.

---

### 9.1.5.3 Perceived and Guaranteed Safety

More than two thirds of the crates do not contain an unsafe abstraction (perceived safety), but less than one third of crates are entirely Safe Rust (guaranteed safety).

38% of crates appear to avoid unsafe, yet contain it in their dependencies.

### 9.1.5.4 Is C the Villain Again?

Calls to C functions are not the majority of the calls to unsafe functions. Implementing some C libraries in Rust, will remove some Unsafe Rust operations, but the majority of unsafe function calls are to Rust functions and Rust intrinsics.

## 9.1.6 How Can Rust Uphold Being a Memory-Safe Language?

The authors suggest changes to the Rust compiler for issuing awareness of hidden unsafety in their Rust code. These are:

- Programmer-assisted automated checks
- Additional tooling to help developers identify uses of unsafe
- More visible code reviews to audit uses of unsafe

More concretely they are proposing:

- A new tag or badge for crates that include Unsafe Rust
- A dependency tree for each library with the crates that use Unsafe Rust clearly marked
- A list of code reviews for any Unsafe Rust.

---

## 9.2 Securing UnSafe Rust Programs with XRust

---

In this paper Peiming Liu, Gang Zhao and Jeff Huang present XRust, "a novel heap allocator that isolates the memory of unsafe Rust from that accessed only in safe Rust, and prevents any cross-region memory corruption". [48]

They claim both high real-world efficiency and effectiveness, as well as support for single- and multi-threaded Rust programs.

XRust dissociates from an approach of transforming unsafe Rust. Rather it tries to integrate unsafe code in safe one, such that it minimizes the amount of collateral damage. In many cases unsafe Rust can corrupt arbitrary data in the whole address space, which can result in severe security leaks. This risk is supposed to be avoidable by XRust's approach of integrating memory-error-prone data.



---


## 10 Conclusion and Future Work

---

C is used nearly everywhere. Its strengths are combining portability, efficient mapping to machine instructions and a stable internal ABI. However, C programmers suffer from the need for manual memory management, making it nearly impossible to write entirely safe C code, given a certain level of project complexity. Rust takes a completely different route, introducing its unique concept of mutability, ownership and lifetimes, which can guarantee memory and thread safety, hence solving the main issues of the C language. Therefore many companies want to move their present C code into Rust. Current state of the art automating solutions for C to Rust transpilation, however, are still premature and nearly unusable in larger and non-trivially complex projects. Looking for different solutions, we found that manual and incremental transitioning of C to Rust is already successfully realized in some projects like *Mozilla Firefox* or *librsvg*. Nonetheless, there is no description, satisfactory detailed documentation or formalization of these successful approaches out there.

In this thesis I formulated a *recipe* for such a manual and incremental porting procedure, allowing C and Rust to co-exist and avoiding "big bang" approaches that require to transpile entire projects at once, on the basis of *librsvg*'s existing incremental transpilation. After narrowing down the minimally required steps from the build part perspective, I introduced language constructs and gave an overview of typical pitfalls and demonstrated how to avoid and overcome them, by successfully applying my recipe on two foreign projects, *calc* and *The Silver Searcher*. Both projects are still compilable and executable after I transplanted one module to Rust, while *The Silver Searcher* is passing each of its provided tests, showing that this incremental porting recipe is working, and is well applicable in larger project settings.

The timeline of this thesis has kept me from applying my recipe on additional projects. Nonetheless, I am optimistic that there are no restrictions in applicability, since we can easily use it to translate smaller units of meaning than modules, like single functions. I see huge optimization potential in the area of automating the process of setting up the



---

recipe's build part. In the future there could be a tool that is able to recognize respective build environment constitutions and automatically select the according build recipe application mechanisms. I can also imagine some optimization for the programming concept recipe. For instance there could be IDE extensions, providing the developer suggestions for translations and type conversion functions, when hovering above certain C constructs or C and Rust interoperability types.

---

## 11 Data Availability

---

In Module Porting to Rust, we looked at snippets of C to Rust transpilations that I performed manually as part of this work. We used these snippets to define fundamentals of an environmental setup for dual C and Rust compilation and linking, as well as for illustration purposes of language features that allow for an incremental transitioning from C to Rust. This chapter provides access to the projects these snippets are taken from and explains their arrangement.

---

### 11.1 GitHub Organization: *IncrementalTransitionOfCCodeIntoRust*

---

The presented projects all can be accessed from the *IncrementalTransitionOfCCodeIntoRust* organization

[33]

where more projects reside in, in which work for this thesis was started and abandoned in favor of the results that will be listed in the following subsections.

#### 11.1.1 lz4 Transpilation

The *lz4* transpilation result with *C2Rust* and manual removal of *Cargo* warnings, discussed in *lz4* and *C2Rust*, can be found in this repository under the `master` branch.

[40]

---

### 11.1.2 librsvg Marker Port Replay Commits

A clean commit history showing nothing but the minimal required steps to reconstruct *librsvg's rsvg-marker port*, discussed in *Replaying librsvg's rsvg-marker Port*, can be accessed in this repository under the branch `marker2rust_again`

[39]

### 11.1.3 calc Transpilation

The results of transpiling *calc's helpers.c*, replacing it with *helpers.rs*, discussed in *calc - Introduction and Data*, can be found in this repository under the branch `calc_rust`.

[38]

### 11.1.4 The Silver Searcher Transpilation

The transpilation of *The Silver Searcher's* filter and ignore functionalities, discussed in *The Silver Searcher - Introduction and Data*, can be found in this repository under the branch `ss2rust_new`.

[41]

---

## Bibliography

---

- [1] *A little C with your Rust: Automatically generating the interface*. rust-embedded. URL: <https://rust-embedded.github.io/book/interoperability/c-with-rust.html#automatically-generating-the-interface> (visited on 09/21/2020).
- [2] *A little Rust with your C*. rust-embedded. URL: <https://rust-embedded.github.io/book/interoperability/rust-with-c.html> (visited on 09/11/2020).
- [3] *A random Rust commit to Firefox by Bert Peers*. Mozilla. URL: [https://bugzilla.mozilla.org/show\\_bug.cgi?id=1603274](https://bugzilla.mozilla.org/show_bug.cgi?id=1603274) (visited on 06/04/2020).
- [4] *ABI*. Wikipedia. URL: [https://en.wikipedia.org/wiki/Application\\_binary\\_interface](https://en.wikipedia.org/wiki/Application_binary_interface) (visited on 09/11/2020).
- [5] *Alternative Representations in Rust*. rust-lang. URL: <https://doc.rust-lang.org/nomicon/other-reprs.html#alternative-representations> (visited on 09/21/2020).
- [6] *assert in Rust*. rust-lang. URL: <https://doc.rust-lang.org/std/macro.assert.html> (visited on 06/18/2020).
- [7] *Assert.h*. Wikipedia. URL: <https://en.wikipedia.org/wiki/Assert.h> (visited on 06/18/2020).
- [8] *bear*. rizsotto. URL: <https://github.com/rizsotto/Bear> (visited on 06/18/2020).
- [9] *C2Rust*. GitHub. URL: <https://github.com/immunant/c2rust> (visited on 03/30/2020).
- [10] *C2Rust Cross-Checking Components*. immunant. URL: <https://c2rust.com/manual/cross-checks/index.html> (visited on 09/30/2020).
- [11] *C2Rust Demonstration*. immunant. URL: <https://c2rust.com/> (visited on 09/30/2020).

- 
- 
- [12] *C2Rust Manual*. immunant. URL: <https://c2rust.com/manual/> (visited on 10/02/2020).
  - [13] *C2Rust Refactoring Tool*. immunant. URL: <https://c2rust.com/manual/c2rust-refactor/> (visited on 09/30/2020).
  - [14] *calc GitHub page*. vivekannan. URL: <https://github.com/vivekannan/calc> (visited on 10/03/2020).
  - [15] *Cargo Toml vs. Cargo Lock*. rust-lang. URL: <https://doc.rust-lang.org/cargo/guide/cargo-toml-vs-cargo-lock.html> (visited on 07/05/2020).
  - [16] *Citrus*. Citrus-rs. URL: <https://gitlab.com/citrus-rs/citrus> (visited on 04/27/2020).
  - [17] *Citrus readme*. Citrus-rs. URL: <https://gitlab.com/citrus-rs/citrus/-/blob/master/README.md> (visited on 05/28/2020).
  - [18] *Closures in Rust*. rust-lang. URL: <https://doc.rust-lang.org/book/ch13-01-closures.html> (visited on 09/23/2020).
  - [19] *Code commenting inside The Silver Searcher's ignore.c on master branch*. Sept. 2020.
  - [20] *codecare*. codecare. URL: <https://www.codecare.de/> (visited on 09/17/2020).
  - [21] *Component Object Model (COM)*. Wikipedia. URL: [https://en.wikipedia.org/wiki/Component\\_Object\\_Model](https://en.wikipedia.org/wiki/Component_Object_Model) (visited on 09/28/2020).
  - [22] *Corrode*. jameysharp. URL: <https://github.com/jameysharp/corrode> (visited on 04/27/2020).
  - [23] *Crate syntex\_syntax*. docs.rs. URL: [https://docs.rs/syntex\\_syntax/0.59.1/syntex\\_syntax/](https://docs.rs/syntex_syntax/0.59.1/syntex_syntax/) (visited on 09/30/2020).
  - [24] *Derive*. rust-lang. URL: <https://doc.rust-lang.org/stable/rust-by-example/trait/derive.html> (visited on 09/21/2020).
  - [25] *Doxygen*. doxygen. URL: <https://www.doxygen.nl/index.html> (visited on 09/21/2020).
  - [26] *enum Result*. rust-lang. URL: <https://doc.rust-lang.org/std/result/enum.Result.html> (visited on 09/12/2020).
  - [27] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. "Is Rust Used Safely by Software Developers?" In: IEEE/ACM 42nd International Conference on Software Engineering (ICSE), 2020.

- 
- 
- [28] *Firefox build instructions*. Mozilla. URL: [https://developer.mozilla.org/en-US/docs/Mozilla/Developer\\_guide/Build\\_Instructions](https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Build_Instructions) (visited on 05/28/2020).
- [29] *full-scene-rasterizer*. jrmuizel. URL: <https://github.com/jrmuizel/full-scene-rasterizer/blob/simple/rasterizer.cc> (visited on 06/11/2020).
- [30] *full-scene-rasterizer Readme*. jrmuizel. URL: <https://github.com/jrmuizel/full-scene-rasterizer/blob/master/README> (visited on 06/11/2020).
- [31] *Function slice\_from\_raw\_parts*. rust-lang. URL: [https://doc.rust-lang.org/beta/std/ptr/fn.slice\\_from\\_raw\\_parts.html](https://doc.rust-lang.org/beta/std/ptr/fn.slice_from_raw_parts.html) (visited on 09/28/2020).
- [32] *gfx-firefox chat conversation with Jeff Muizelaar*. June 2020.
- [33] *GitHub organization: IncrementalTransitionOfCCCodeIntoRust*. GitHub. URL: <https://github.com/IncrementalTransitionOfCCCodeIntoRust> (visited on 09/29/2020).
- [34] *golang*. Google. URL: <https://golang.org/> (visited on 04/01/2020).
- [35] *harfbuzz*. harfbuzz. URL: <https://github.com/harfbuzz/harfbuzz> (visited on 06/11/2020).
- [36] *harfbuzz Readme*. harfbuzz. URL: <https://github.com/harfbuzz/harfbuzz/blob/master/README.md> (visited on 06/11/2020).
- [37] *Immunant Inc*. immunant. URL: <https://immunant.com/> (visited on 10/04/2020).
- [38] *IncrementalTransitionOfCCCodeIntoRust: calc*. GitHub. URL: <https://github.com/IncrementalTransitionOfCCCodeIntoRust/calc> (visited on 09/29/2020).
- [39] *IncrementalTransitionOfCCCodeIntoRust: libsvg\_c*. GitHub. URL: [https://github.com/IncrementalTransitionOfCCCodeIntoRust/libsvg\\_c](https://github.com/IncrementalTransitionOfCCCodeIntoRust/libsvg_c) (visited on 09/29/2020).
- [40] *IncrementalTransitionOfCCCodeIntoRust: lz4\_rust*. GitHub. URL: [https://github.com/IncrementalTransitionOfCCCodeIntoRust/lz4\\_rust](https://github.com/IncrementalTransitionOfCCCodeIntoRust/lz4_rust) (visited on 09/29/2020).
- [41] *IncrementalTransitionOfCCCodeIntoRust: The Silver Searcher*. GitHub. URL: [https://github.com/IncrementalTransitionOfCCCodeIntoRust/the\\_silver\\_searcher](https://github.com/IncrementalTransitionOfCCCodeIntoRust/the_silver_searcher) (visited on 09/29/2020).
- [42] *libexpat*. libexpat. URL: <https://github.com/libexpat/libexpat> (visited on 06/11/2020).

- 
- [43] *libexpat Readme*. libexpat. URL: <https://github.com/libexpat/libexpat/blob/master/README.md> (visited on 06/11/2020).
- [44] *librsvg*. GNOME. URL: <https://github.com/GNOME/librsvg> (visited on 06/11/2020).
- [45] *librsvg: Initial Rust commit*. GNOME. URL: <https://github.com/GNOME/librsvg/commit/f27a8c908ac23f5b7560d2279acec44e41b91a25> (visited on 06/11/2020).
- [46] *librsvg: marker.rs: Fully implement markers in Rust. rsvg-marker.c is gone!* GNOME. URL: <https://github.com/GNOME/librsvg/commit/6dd4ffd62f9d68799d1b2372a0542> (visited on 06/11/2020).
- [47] *Linux Kernel Programmiersprache*. Wikipedia. URL: [https://de.wikipedia.org/wiki/Linux\\_\(Kernel\)#Programmiersprache](https://de.wikipedia.org/wiki/Linux_(Kernel)#Programmiersprache) (visited on 09/18/2020).
- [48] P. Liu, G. Zhao, and J. Huang. “Securing Unsafe Rust Programs with X Rust”. In: IEEE/ACM 42nd International Conference on Software Engineering (ICSE), 2020.
- [49] *loc: GitHub Readme.md*. cgag. URL: <https://github.com/cgag/loc> (visited on 10/03/2020).
- [50] *locka99.gitbooks.io: Types*. gitbooks.io. URL: [https://locka99.gitbooks.io/a-guide-to-porting-c-to-rust/content/features\\_of\\_rust/types.html](https://locka99.gitbooks.io/a-guide-to-porting-c-to-rust/content/features_of_rust/types.html) (visited on 10/04/2020).
- [51] *lz4 Readme*. lz4. URL: <https://github.com/lz4/lz4> (visited on 06/18/2020).
- [52] *lz4 Readme*. lz4. URL: <https://github.com/lz4/lz4/blob/dev/README.md> (visited on 06/18/2020).
- [53] *macro lazy\_static*. docs.rs. URL: [https://docs.rs/lazy%7B%5C\\_%7Dstatic/1.4.0/lazy%7B%5C\\_%7Dstatic/](https://docs.rs/lazy%7B%5C_%7Dstatic/1.4.0/lazy%7B%5C_%7Dstatic/) (visited on 09/13/2020).
- [54] *Method collect*. rust-lang. URL: <https://doc.rust-lang.org/nightly/std/iter/trait.Iterator.html#method.collect> (visited on 09/23/2020).
- [55] *Method extension*. rust-lang. URL: <https://doc.rust-lang.org/std/path/struct.Path.html#method.extension> (visited on 09/22/2020).
- [56] *Method is\_null*. rust-lang. URL: [https://doc.rust-lang.org/std/primitive.pointer.html#method.is\\_null-1](https://doc.rust-lang.org/std/primitive.pointer.html#method.is_null-1) (visited on 09/28/2020).
- [57] *Method iter*. rust-lang. URL: <https://doc.rust-lang.org/std/iter/index.html> (visited on 09/23/2020).



- 
- 
- [58] *Method position*. rust-lang. URL: <https://doc.rust-lang.org/std/iter/trait.Iterator.html#method.position> (visited on 09/23/2020).
- [59] *method unwrap*. rust-lang. URL: <https://doc.rust-lang.org/std/result/enum.Result.html#method.unwrap> (visited on 09/12/2020).
- [60] *Method: as\_ptr*. doc.rust-lang.org. URL: [https://doc.rust-lang.org/std/primitive.slice.html#method.as\\_ptr](https://doc.rust-lang.org/std/primitive.slice.html#method.as_ptr) (visited on 10/04/2020).
- [61] *Mozilla Description of Components: Core*. Mozilla. URL: <https://bugzilla.mozilla.org/describecomponents.cgi?product=Core> (visited on 10/01/2020).
- [62] *Pointer Types*. rust-lang. URL: <https://doc.rust-lang.org/reference/types/pointer.html> (visited on 09/12/2020).
- [63] *raqote*. jrmuizel. URL: <https://github.com/jrmuizel/raqote/blob/master/src/rasterizer.rs> (visited on 06/11/2020).
- [64] *rexpats*. immunant. URL: <https://github.com/immunant/rexpats> (visited on 06/11/2020).
- [65] *rexpats Readme*. immunant. URL: <https://github.com/immunant/rexpats/blob/master/README.md> (visited on 06/11/2020).
- [66] Karl Rikte. “Using Rust as a Complement to C for Embedded Systems Software Development (A Study Performed Porting a Linux Daemon)”. MA thesis. 2018.
- [67] *Rust - Modules*. tutorialspoint. URL: [https://www.tutorialspoint.com/rust/rust\\_modules.htm](https://www.tutorialspoint.com/rust/rust_modules.htm) (visited on 09/10/2020).
- [68] *Rust 2020: Lessons learned by transpiling C to Rust*. URL: <https://immunant.com/blog/2019/11/rust2020/> (visited on 03/30/2020).
- [69] *Rust Build Scripts*. rust-lang. URL: <https://doc.rust-lang.org/cargo/reference/build-scripts.html#build-scripts> (visited on 09/21/2020).
- [70] *Rust is not a good C replacement*. URL: <https://drewdevault.com/2019/03/25/Rust-is-not-a-good-C-replacement.html> (visited on 03/30/2020).
- [71] *rust-bindgen*. rust-lang. URL: <https://github.com/rust-lang/rust-bindgen> (visited on 09/21/2020).
- [72] *Rust’s cty crate*. crates.io. URL: <https://crates.io/crates/cty> (visited on 09/11/2020).
- [73] *Rust’s libc crate*. gitbooks.io. URL: <https://docs.rs/libc/0.2.78/libc/> (visited on 10/04/2020).

- 
- 
- [74] *rustconf2018*. RustConf. URL: <https://www.youtube.com/watch?v=WEsR0Vv7jhg> (visited on 04/27/2020).
- [75] *rustibuzz*. RazrFalcon. URL: <https://github.com/RazrFalcon/rustybuzz> (visited on 06/11/2020).
- [76] *Security: Microsoft baut COM Bibliothek in Rust*. Golem. URL: <https://www.golem.de/news/security-microsoft-baut-com-bibliothek-in-rust-1910-144358.html> (visited on 03/30/2020).
- [77] *Separation of Concerns for Binary Projects*. rust-lang. URL: <https://doc.rust-lang.org/book/ch12-03-improving-error-handling-and-modularity.html?highlight=lib.rs#separation-of-concerns-for-binary-projects> (visited on 09/10/2020).
- [78] *Source-to-source compiler*. Wikimedia Foundation, Inc. URL: [https://en.wikipedia.org/wiki/Source-to-source\\_compiler](https://en.wikipedia.org/wiki/Source-to-source_compiler) (visited on 05/28/2020).
- [79] *Struct chars*. rust-lang. URL: <https://doc.rust-lang.org/std/str/struct.Chars.html> (visited on 09/23/2020).
- [80] *struct CStr*. rust-lang. URL: <https://doc.rust-lang.org/std/ffi/struct.CStr.html> (visited on 09/12/2020).
- [81] *struct CString*. rust-lang. URL: <https://doc.rust-lang.org/std/ffi/struct.CString.html> (visited on 09/12/2020).
- [82] *Struct filter*. rust-lang. URL: <https://doc.rust-lang.org/std/iter/trait.Iterator.html> (visited on 09/23/2020).
- [83] *The Cargo Book*. doc.rust-lang.org. URL: <https://doc.rust-lang.org/cargo/> (visited on 10/04/2020).
- [84] *The Rust community's crate registry*. crates.io. URL: <https://crates.io/> (visited on 10/04/2020).
- [85] *The Rust Programming Language: Unsafe Rust*. Mozilla. URL: <https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html> (visited on 04/01/2020).
- [86] *The Rust Reference: Crates and Source Files*. doc.rust-lang.org. URL: <https://doc.rust-lang.org/reference/crates-and-source-files.html> (visited on 10/04/2020).
- [87] *The Rust Reference: Extern crate declarations*. doc.rust-lang.org. URL: <https://doc.rust-lang.org/reference/items/extern-crates.html> (visited on 10/04/2020).

- 
- 
- [88] *The Rust Reference: Functions*. doc.rust-lang.org. URL: <https://doc.rust-lang.org/reference/items/functions.html> (visited on 10/04/2020).
- [89] *The Rust Reference: Modules*. doc.rust-lang.org. URL: <https://doc.rust-lang.org/reference/items/modules.html> (visited on 10/04/2020).
- [90] *The Rust Reference: Mutability*. doc.rust-lang.org. URL: <https://doc.rust-lang.org/reference/expressions.html?highlight=mutability#mutability> (visited on 10/04/2020).
- [91] *The Rust Reference: Ownership*. doc.rust-lang.org. URL: <https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html> (visited on 10/04/2020).
- [92] *The Rust Reference: Place Expressions and Value Expressions*. doc.rust-lang.org. URL: <https://doc.rust-lang.org/reference/expressions.html?highlight=place,expression#place-expressions-and-value-expressions> (visited on 10/04/2020).
- [93] *The Rust Reference: References and Borrowing*. doc.rust-lang.org. URL: <https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html> (visited on 10/04/2020).
- [94] *The Rust Reference: Validating References with Lifetimes*. doc.rust-lang.org. URL: <https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html> (visited on 10/04/2020).
- [95] *The Silver Searcher: GitHub Readme.md*. ggreer. URL: [https://github.com/ggreer/the\\_silver\\_searcher](https://github.com/ggreer/the_silver_searcher) (visited on 09/14/2020).
- [96] *Vivek Kannan at GitHub*. vivekannan. URL: <https://github.com/vivekannan> (visited on 09/12/2020).
- [97] *Why Discord is switching from Go to Rust*. Discord. URL: <https://blog.discordapp.com/why-discord-is-switching-from-go-to-rust-a190bbca2b1f> (visited on 03/30/2020).
- [98] *Wikipedia: Garbage Collection (computer science)*. Wikipedia. URL: [https://en.wikipedia.org/wiki/Garbage\\_collection\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)) (visited on 10/04/2020).